LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TH-226455

An Analysis Framework Addressing the Scale and Legibility of Large Scientific Data Sets

H. R. Childs

November 30, 2006

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Hank R. Childs
December 2006
Computer Science

An Analysis Framework Addressing the Scale and Legibility of Large Scientific Data Sets

Abstract

Much of the previous work in the large data visualization area has solely focused on handling the scale of the data. This task is clearly a great challenge and necessary, but it is not sufficient. Applying standard visualization techniques to large scale data sets often creates complicated pictures where meaningful trends are lost. A second challenge, then, is to also provide algorithms that simplify what an analyst must understand, using either visual or quantitative means. This challenge can be summarized as improving the legibility or reducing the complexity of massive data sets. Fully meeting both of these challenges is the work of many, many PhD dissertations. In *this* dissertation, we describe some new techniques to address both the scale and legibility challenges, in hope of contributing to the larger solution.

In addition to our assumption of simultaneously addressing both scale and legibility, we add an additional requirement that the solutions considered fit well within an interoperable framework for diverse algorithms, because a large suite of algorithms is often necessary to fully understand complex data sets.

For scale, we present a general architecture for handling large data, as well as details of a contract-based system for integrating advanced optimizations into a data flow network design. We also describe techniques for volume rendering and performing comparisons at the extreme scale. For legibility, we present several techniques. Most noteworthy are equivalence class functions, a technique to drive visualizations using statistical methods, and line-scan based techniques for characterizing shape.

An Analysis Framework Addressing the Scale and Legibility of Large Scientific Data Sets

By

HANK R. CHILDS
B.S. (University of California, Davis) 1999

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Committee in charge

2006

Acknowledgments

I thank my family for the support they have provided. My wife, Carissa, has never complained about the hundreds to thousands of hours I have spent throughout this process. Without her unfailing support and her willingness to put my dreams and goals first, this would not have been possible.

I thank my adviser, Nelson Max, for his guidance, expertise, and support through this process. He worked tirelessly on my behalf and I am appreciative. I also thank Ken Joy for offering expertise and mentoring, even though I was not one of his students. Finally, I thank the third member of my thesis committee, Mark Duchaineau, for his expertise and effort, as well as his friendship and career advice throughout the years.

Since the work regarding the VisIt project has shaped much of my dissertation, I thank the VisIt team members who I have worked so happily with for the last five to seven years: Eric Brugger, Kathleen Bonnell, Mark Miller, Jeremy Meredith, Sean Ahern, and Brad Whitlock. Many of the ideas presented in this document came about as a result of conversations of those folks and I am indebted to them.

I thank the countless co-workers who have offered support, ranging from encouraging comments to specific advice for navigating through a graduate degree. Specifically, I call out Evi Dube, Barb Kornblum, and Tom Adams.

I thank all of the co-authors for the papers I have participated in. Their contributions have directly made this dissertation better. They include: Sean Ahern, Kathleen Bonnell, Eric Brugger, John Clyne, Mark Duchaineau, Ken Joy, Ed Kokko, Kwan-Liu Ma, Nelson Max, Jeremy Meredith, Mark Miller, George Ostrochov, and Brad Whitlock.

I thank the members of my qualifying examination: Mark Duchaineau, Ken Joy, Nelson Max, Kwan-Liu Ma, and Valerio Pascucci. Reviewing this work is tedious and I am honored that such a distinguised group has done so. Again, I thank the thesis committee: Mark Duchaineau, Ken Joy, and Nelson Max.

Thank you all!

Contents

List of Figures

Abstract

Much of the previous work in the large data visualization area has solely focused on handling the scale of the data. This task is clearly a great challenge and necessary, but it is not sufficient. Applying standard visualization techniques to large scale data sets often creates complicated pictures where meaningful trends are lost. A second challenge, then, is to also provide algorithms that simplify what an analyst must understand, using either visual or quantitative means. This challenge can be summarized as improving the legibility or reducing the complexity of massive data sets. Fully meeting both of these challenges is the work of many, many PhD dissertations. In *this* dissertation, we describe some new techniques to address both the scale and legibility challenges, in hope of contributing to the larger solution.

In addition to our assumption of simultaneously addressing both scale and legibility, we add an additional requirement that the solutions considered fit well within an interoperable framework for diverse algorithms, because a large suite of algorithms is often necessary to fully understand complex data sets.

For scale, we present a general architecture for handling large data, as well as details of a contract-based system for integrating advanced optimizations into a data flow network design. We also describe techniques for volume rendering and performing comparisons at the extreme scale. For legibility, we present several techniques. Most noteworthy are equivalence class functions, a technique to drive visualizations using statistical methods, and line-scan based techniques for characterizing shape.

Chapter 1

Introduction

1.1 Motivation

Supercomputing, once again, is a booming industry. The field was jumpstarted in 1995 by the Accelerated Strategic Computing Initiative (ASCI), which has gone on to dominate the Top500 list[76] for the last ten years. But supercomputing centers commensurate to or exceeding ASCI have emerged. These efforts promise even larger machines and even higher resolution simulations. Examples include the Earth Simulator Center in Japan, which produced the "Earth Simulator" machine, the Commissariat a l'Energie Atomique (CEA) in France, which produced the "Tera-10" machine, and the SciDAC effort (**Sci**entific **D**iscovery through **A**dvanced **C**omputing) through the Department of Energy's Office of Science. With countless other large supercomputing efforts either starting or well underway, the issues of visualizing and analyzing massive data sets has never been more important.

The sizes of the data sets being produced by these computers are staggering. A single simulation can now operate on billions of elements and produce terabytes of data. Petabyte scale simulations are on the near horizon. With each jump in compute power (and corresponding jump in mesh resolution), the amount of data to comprehend also jumps. The data already well exceeds what an analyst can explore without sophisticated techniques, and the problem will only worsen into the future.

The purpose of *post-processing tools* is to enable analysts to increase their understanding of these data sets. They do this by employing a varied suite of visual and

quantitative techniques.

Massive data sets pose *two* incredible challenges to post-processing tools. The first challenge is to seamlessly handle the scale of the data. Further, the goal is not to simply process this data, but to process this data interactively.

The second challenge is perhaps even more difficult than the first. Large scale simulations normally model extremely complex phenomena. When processing this data, standard visualization techniques often produce pictures that exceed what the human brain's processing system can fully comprehend. So the challenge is to improve how this information is conveyed, with the ultimate goal being to increase data understanding. Throughout this paper, we refer to this challenge as *improving legibility* at the large scale. The intent of the phrase "legibility" is to focus on making underlying phenomena more apparent, as opposed to connotations of filtering out noisy data.

To further illustrate these two goals, consider the following example. A recent simulation on LLNL's Blue Gene/L machine modeled a Rayleigh Taylor instability, where light and heavy fluids mix. The simulation contained twenty-seven billion mesh elements at each time slice. Details of the simulation can be found in [13]. Simply applying standard visualization techniques to this data required an architecture that both operated in parallel and incorporated sophisticated optimizations. Making figure 1.1 required technology to respond to the first challenge, addressing the large scale of the data. However, the second challenge still looms large. Consider figure 1.2. It is of a predecessor run by the same simulation code that uses a mere one billion elements. By looking at these two pictures, can you infer differences between the two simulations? Even if you are an expert analyst in the field of fluid dynamics, the answer is almost certainly "no". Although the pictures are informative, they are complex enough to exceed the limits of what we, as humans, can meaningly infer with our visual processing systems. So we must still provide techniques to improve the legibility of these data sets. In this case, an important metric to characterize this type of simulation is the measure of complexity of the mixing region between the two fluids. This is done by calculating a boundary between the fluids (chosen as an isovalue for some intermediate density) and then calculating the complexity as the surface area of the boundary. A plot comparing the surface areas of the boundary is shown in figure 1.3.

The twenty-seven billion element simulation demonstrates a substantially different mixing behavior (seen with the bump that flattens out early in the simulation) that could never be inferred by looking at figures 1.1 and 1.2 or even a movie of these figures animated over time.

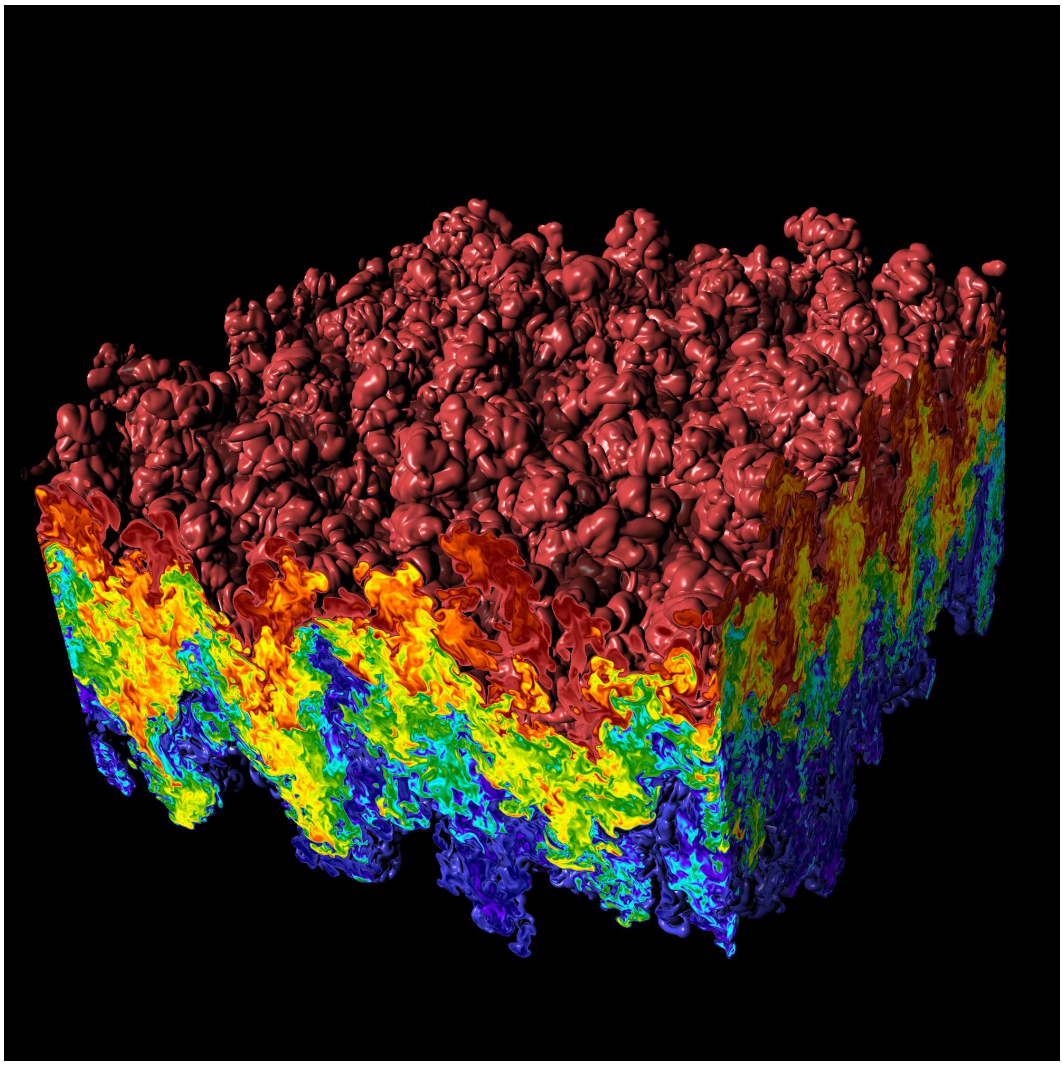

Figure 1.1: A rendering of a 27-billion element Rayleigh Taylor instability

1.2 Research Goals

Throughout this dissertation, we will refer to the concept of a richly featured tool. A richly featured tool is a tool for processing general data, i.e unstructured and

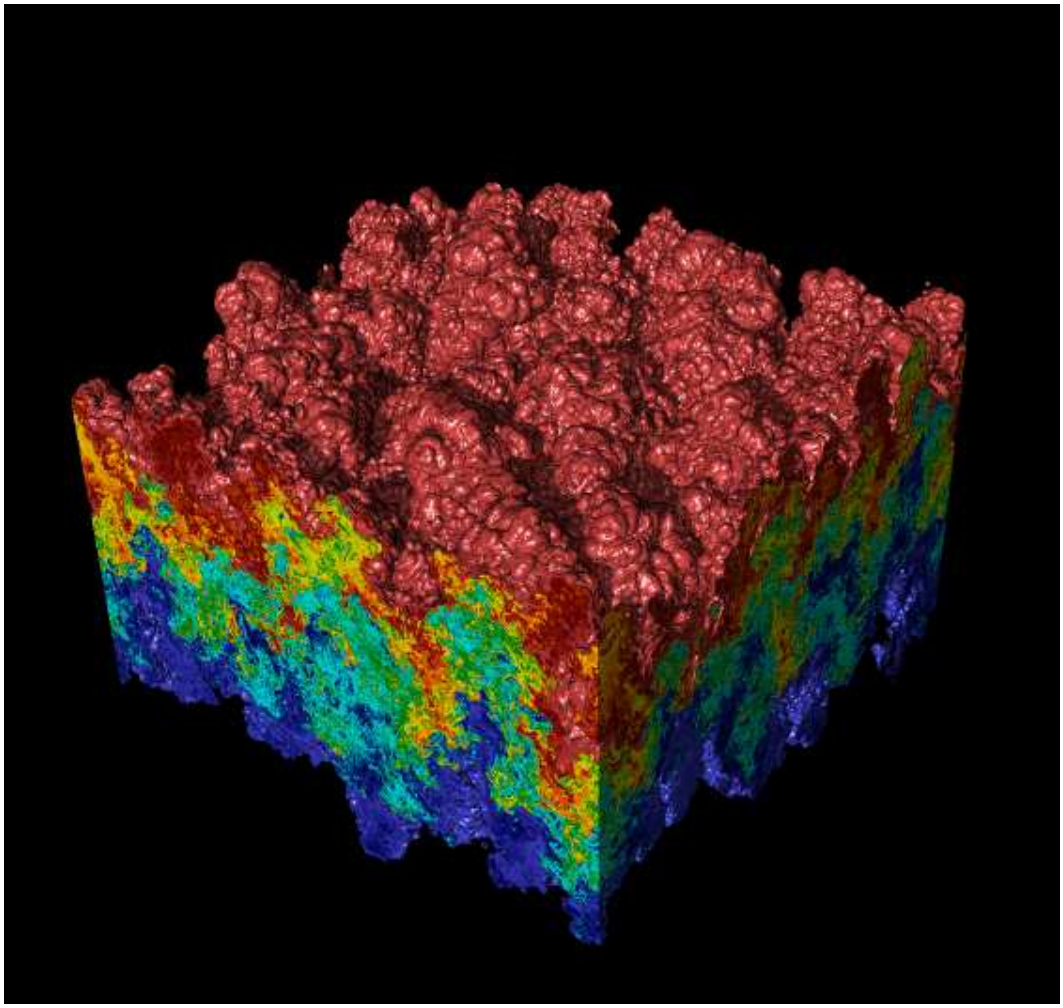

Figure 1.2: A rendering of a 1-billion element Rayleigh Taylor instability

structured meshes, scalars, vectors, tensors, etc. Richly featured tools also contain hundreds of algorithms, many of which can be composed together to meet dynamic analysis needs.

We strongly believe that richly featured tools with interoperable feature sets are superior to specialized applications when it comes to performing conclusive data analysis. Interoperability enables techniques such as derived quantity generation, data manipulation, plotting variations, and quantitative techniques to all be included in the same analysis.

This dissertation assumes a requirement that the techniques described can be deployed inside a richly featured tool. Of course, by choosing richly featured tools as a target, there is a key drawback. Many specialized applications are designed around optimizing the workflow for a handful of operations. For example, a tool that does only

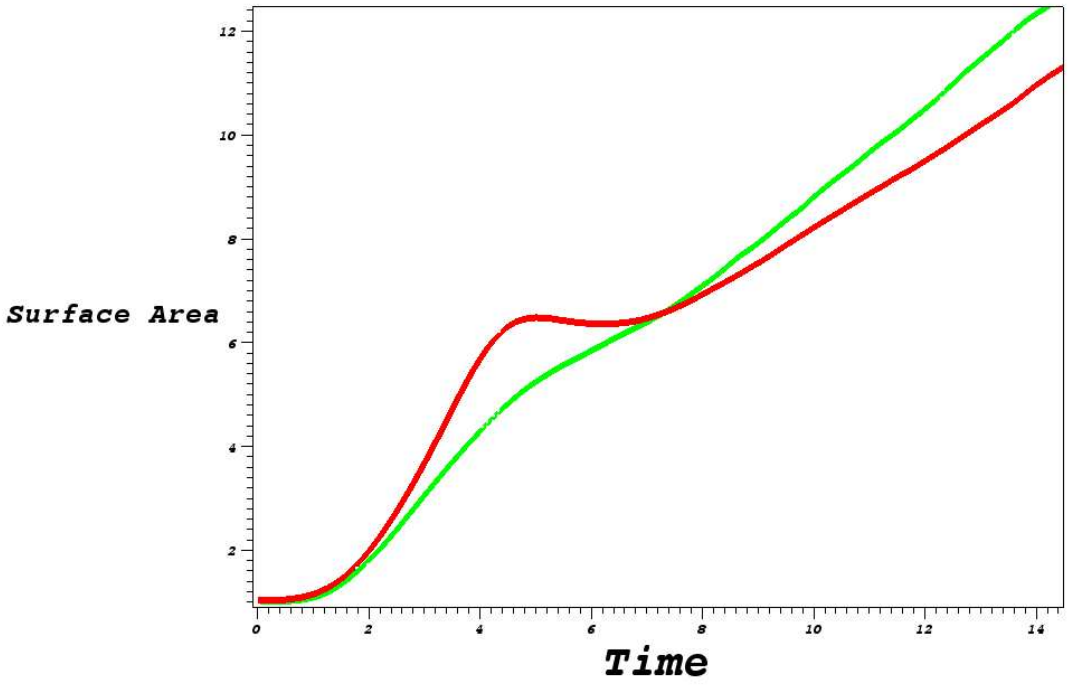

Figure 1.3: Here we plot the surface area of the boundary for the twenty-seven billion element calculation (red) and the one billion element calculation (green).

volume rendering may build its I/O, data management, communication, and rendering algorithms around a specific volume rendering workflow. With a richly featured tool, this is not practical because of the potential interactions between the operations. A richly featured tool must incorporate general strategies for I/O, data management, communication, and rendering that support all its operations.

The challenges of visualizing and analyzing large-scale data are complex. This dissertation offers no single panacea. Instead, the principle contribution of this dissertation is two-fold. The first is to describe an architecture that is suitable for production-level analysis tools with rich, interoperable features sets and that is capable of incorportating sophisticated optimizations. The second is to contribute some advanced techniques, either in addressing scale or improving legibility, that fit within such a framework.

Chapters 2, 3, 4, and 5 all address handling the scale of large data sets. Chapter 2 gives an overview of the architecture, while chapter 3 details how to apply optimizations in such an architecture. Chapter 4 provides an algorithm for volume rendering large data sets efficiently within a general architecture. And chapter 5 describes how to perform comparisons at the extreme scale.

Chapters 6, 7, and 8 all address improving the legibility of large data sets. Chapter 6 outlines basic quantitative techniques and their utility. Chapter 7 provides a methodology for combining statistical operators with standard visualization techniques. Finally, chapter 8 shows how line-scan based techniques can be employed for shape characterization.

Part I

Handling the scale of large data

Chapter 2

An Architecture for Large Scale Data And Richly Featured Tools

In this chapter[1], we propose a client-server design that parallelizes all data processing. The design is very flexible and is well suited to the requirements of richly featured tools: support for a wide array of operations, support for many data models, and straight forward extensibility. This design has been implemented in VisIt, a visualization and analysis tool.

The general data management strategy for the parallel server can essentially be described as a "Scatter-Gather" algorithm. The process can be characterized in three steps:

1. I/O (scatter)
2. Data Flow Networks (processing)
3. Rendering (gather)

In a traditional Scatter-Gather algorithm, the Scatter phase involves partitioning and distributing data across processors using parallel communication. For our purposes, the data is effectively pre-partitioned (by the simulation) before the post-processing tool reads it. So, the data is read and distributed in chunks without any communication between processors during the Scatter phase.

[1]Much of the text from this chapter comes from [19], which was a collaboration between Hank Childs and Mark Miller

In the following subsections, we will further describe the proposed architecture in more depth. The purpose of this overview is to provide background to the reader, although we comment that the I/O model is extremely flexible and tailored to the diverse requirements placed on a general tool when it comes to reading data from a plethora of simulation codes.

2.1 Related Work

Data flow network systems have been implemented in many systems, for example VTK [67], OpenDX [1], and AVS [78]. The distinguishing feature of the proposed data flow network system is the "contract" extension that enables optimizations to be applied adaptively. This capability will be described in more detail in chapter 3.

There are many richly-featured parallel visualization tools that perform data reduction in parallel followed by a combined rendering stage, although these tools frequently struggle with operating on the data in its original form, supporting collective communication, and/or incorporating advanced optimizations. Examples of these are EnSight [21], ParaView [39], PV3 [32] and FieldView [40]. The system that is described in the rest of this chapter is able to keep data in its original form, while also supporting collective communication and incorporating advanced optimizations.

The data types supported by richly featured tools are varied, including unstructured, curvilinear, rectilinear, AMR, and point-based meshes, as scalar, vector, and tensor data, spectral data (i.e. for energy groups), and material volume fractions. Support for some of these non-standard data types is itself a research topic, including material interface reconstruction algorithms from Bonnell et al. [10, 11], contouring algorithms for AMR meshes from Weber et al. [82], and volume rendering of unstructured and curvilinear meshes from Max et al. [46] and Ma and Crockett [44].

Our design is based upon a client/server design where the server is parallelized. The motivations for this design are similar to those described in many papers, for example those of Bethel et al. [9] and Ma and Camp [43] – processing data in parallel on the machine it was generated on, while achieving interactive rendering rates. However, note that the

visualization and analysis jobs typically receive only a small fraction of the compute power than the simulations code receive, so the job of processing this data is more difficult. Finally, it should be noted that the rendering paradigm outlined in subsection 3.3 is different from that described in [9] and very different from [43], which is optimized for time-varying volume rendering.

The proposed server can take two forms. One form is as a stand-alone process that accesses data through the file system. The other form is as a library that is linked into an application code. This second use case allows for real-time monitoring and potentially even computational steering. Haimes's PV3 system [31], and the SCIRun system of Parker, Johnson, and others [35, 51] took a similar approach, but have not addressed similar scale. Pascucci et al. [59] addressed real-time monitoring at large scale, but their solution required separate compute power. Their solution focused on making minimal impact on the simulation code, rather than leveraging the power of the simulation's compute platform. Others have taken proprietary toolkits and embedded them in a simulation environment, for example Allen et al. [4] putting AMIRA into CACTUS, and the CUMULVS system [29] on top of AVS.

Yu, Ma, and Welling present I/O strategies for time-varying data in a distributed computing environment [87]. Their approach is compelling, but the overlapping of I/O and processing creates difficulties in the setting of a richly featured tool.

2.2 I/O

Each processor of the parallel server reads a portion of the input data set. This is the mechanism that "scatters" the data set across its server. The key question during the I/O phase is how to assign portions of the input data set to the processors of the server. Before this question can be answered, first consider how this data is inputted.

Data can be obtained from a simulation in one of two fundamentally different ways. The first, most common use case is to run as a standalone application where the data is read from secondary storage. That is, the data is read from files in a filesystem. The second use case is to run as a library linked into a simulation code and read data directly

from primary memory. That is, the data is read directly from the memory of a running simulation. Note that the data may need to be re-formatted to match the server's input requirements leading to additional memory overhead. Nonetheless, the memory overhead is often worth it, given the potential performance gains of bypassing the filesystem and the advantage of reducing storage space requirements.

Next, when the server processes a portion of the data set, the input data set must be partitioned and distributed across the server's processors. The question becomes: what restrictions does the input data's layout impose on partitioning and distributing that data for parallel processing? There are three scenarios:

1. The data is being read from file(s) and the underlying I/O infrastructure only supports a partitioning scheme fixed by the simulation when the file(s) were created.

2. The data is being read from file(s) and the underlying I/O infrastructure supports re-partitioning during read.

3. The data is being read from the memory of a running simulation, so the partitioning scheme is fixed by how it is stored in the memory of the simulation.

In all cases, the simulation has already partitioned the data at least one time; this partitioning is a one-to-one correspondence between each processor of the parallel simulation code and the elements that that processor operates on[2]. The grouping of elements that belong to a single processor is often referred to as a domain. Examples of I/O libraries that support a partitioning scheme fixed by the simulation at the time the files are created are Silo [63] and Exodus [66]. While it is conceivable to alter this partitioning by operating on sub-portions of domains, there is nothing to be gained unless the underlying I/O library supports partial I/O (Silo and Exodus do not). On the other hand, some formats permit re-partitioning of the data when it is read. Examples are described in ViSUS [58] and SAF [52].

So what partitioning should be used? For cases #1 and #3, in our implementation (VisIt), we simply accept the imposed partitioning, because the costs of re-partitioning

[2]This statement only applies to real elements. Sometimes elements along the boundary are replicated on other processors as ghost elements, for interpolation purposes. These ghost elements can not be described using a one-to-one correspondence.

that data typically exceeds the benefits. For case #2, we allow the data reader to perform the partitioning, as the reader often has more domain knowledge about what types of partitionings are appropriate.

We will refer to a "chunk" as a group of elements that are processed all at once. Note that if the partitioning of the data is fixed based on domain decomposition, then chunks correspond directly to domains. Furthermore, visualization tools often get many less processors to work with than the simulation codes, so, in this case, there will likely be more chunks than there are processors on the server. So the server must support chunk overloading, where each processor of the server operates on multiple chunks. Note that an alternate strategy would be to combine all of the chunks on one processor into "super-chunks". This strategy often breaks down in the context of a richly featured tool. For example, there is often no way to combine chunks that are rectilinear in nature into one "super-chunk" that is also rectilinear.

Regardless of the form of the input, the data readers must be able identify chunks and read those chunks for further processing. When the input data is read from the memory of a simulation, the chunk is simply the elements that the simulation has on that processor. When the data is read from a file with a fixed partitioning, we must do chunk overloading with the existing domains defined by the file. And when the input data is read from a file that supports repartitioning, the data reader can be made smart enough to dynamically decompose the data so that each processor has an approximately equal-work chunk.

2.3 Data Flow Networks

Our goal was to build a system with many interoperable features, so it was an obvious choice to use data flow networks ([1, 67, 78]). Again, we extended the basic data flow network concept with a contract design. This will be discussed further in chapter 3. Further, we note that many of the modules in VisIt's data flow network implementation perform common visualization operations using a third party library (VTK [67]).

When the client asks the server to perform a calculation, each processor of the server sets up an identical data flow network. They only differ in the list of chunks from

the data set that they process. The majority of visualization operations are "embarrassingly parallel" – the processing can occur in parallel with no communication. For these operations the only concern is artifacts which can occur along the boundaries of a chunk. These artifacts are typically addressed through the use of ghost data. A small portion of visualization operations, however, are not embarrassingly parallel and do require collective communication between the processors. Both classes of algorithms are supported and ghost data is supported as well. Further description of these concepts can be found in 3.4.2 and 3.4.3, respectively.

2.4 Rendering

When rendering surfaces, we use an adaptive rendering strategy based on the size of a surface to be rendered. If the surface to be rendered contains a small number of geometric primitives, then the server will send that surface to the client to be rendered locally using graphics hardware. However, this approach is inherently non-scalable. So an alternate, backup strategy is needed for large surfaces. In VisIt, the strategy is a parallel rendering mode that will accommodate surfaces made up of large numbers of geometric primitives. It has a user-definable heuristic for how large a surface can be before it thinks it will overwhelm the client. Alternatively, users can explicitly choose which rendering mode they would like to use.

Many papers describe techniques for rendering in parallel. See [53] and [55] for examples. The parallel rendering algorithms described here are not new. However, they are versatile and integrate seamlessly the client/server architecture. We describe them here for completeness.

In parallel, each processor's data flow network produces a portion of the data set to be rendered. Each processor then renders its portion of the data independently, either by using graphics cards (if available), or by using Mesa [60], a software implementation of the OpenGL interface. No attempt is made to re-distribute the data in image space. So, each processor's output image is equal in size to the intended final image and consists of both color and depth values. The individual images are composited in parallel to produce

the final image (see figure 2.1).

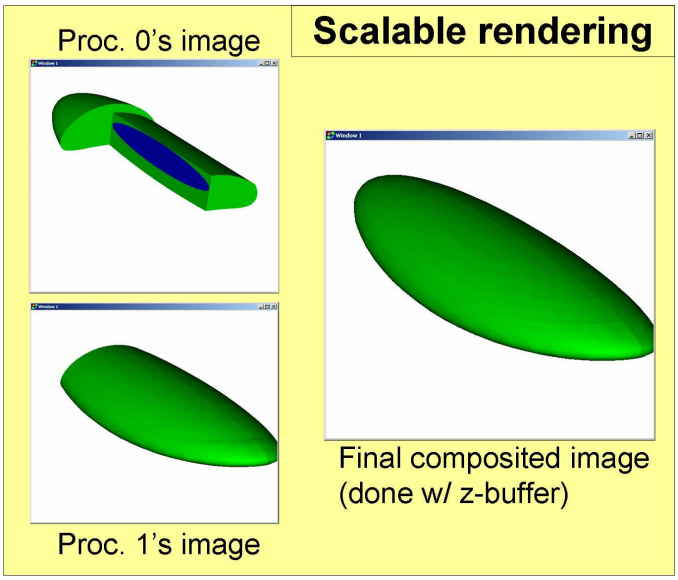

Figure 2.1: Scalable rendering is achieved by having each processor render its own portion of the larger data set and then composite using the z-buffer.

Accommodations are made for renderings including semi-transparent surfaces or volumes. In this case, rendering is performed in multiple passes. Opaque geometry, if any, is rendered and composited in the first pass to provide a background image for subsequent passes. During the next pass, data is re-distributed as appropriate for the particular rendering technique (semi-transparent geometry or ray-cast volume rendering[3]) and combined with the background image from the first pass (which includes depth information). If shadows are desired, another rendering pass is made for shadow maps [84].

After rendering is completed, the image is compressed and shipped to the client. When multiple parallel servers are used, each server ships color and depth values to the client and a final compositing step is performed there. Otherwise, only color values are shipped to the client.

[3]Note that either of these techniques alone can be solved with an additional rendering pass, but performing both in rendering is substantially harder. Here, we refer to the situation where it is one or the other.

Chapter 3

Enabling Advanced Techniques for Large Data

The architecture presented in chapter 2 targets a richly featured tools with interoperable features. That architecture results lends itself to a "brute force" approach, where data is processed naively. In this chapter[1], we will discuss how a simple and elegant extension to the data flow network design, contracts, can allow sophisticated optimizations to be adaptively employed in the context of chapter 2's architecture. The result is an architecture that is both general and efficient. Contracts enable each component of the data flow network to modify the set of optimizations used. In addition, they allow for new components to be accommodated gracefully within a data flow network system.

3.1 Related Work

The concept of reading in only portions of a larger data set (see 3.4.1) has been well discussed, for example by Chiang, et al. [16] and Pascucci, et al. [58]. The concepts described in this chapter are how to incorporate optimizations such as these within a richly featured tool.

The execution model discussed in 3.4.2, *streaming*, maps well to out-of-core pro-

[1]Much of the text from this chapter comes from [17], which was a collaboration between Hank Childs, Eric Brugger, Kathleen Bonnell, Jeremy Meredith, Mark Miller, Brad Whitlock, and Nelson Max

cessing. Many out-of-core algorithms are summarized by Silva et al. in [71]. In addition, Ahrens et al. [3] give an overview of a parallel streaming architecture. In this chapter, the discussion will address when streaming is a viable technique and how the contract-based system enables automatic selection of the best execution model for a given data flow network.

Ghost elements are often created by the simulation code and stored with the rest of the data. The advantages of utilizing ghost elements to avoid artifacts at domain boundaries (see 3.4.3) were discussed in [3]. In this chapter, we propose that the post-processing tool be used to generate ghost data when ghost data is not available in the input data. Further, we discuss the factors that require when and what type of ghost data should be generated, as well as a system that can incorporate these factors (i.e. contracts).

3.2 Integrating Contracts with Data Flow Networks

Contracts extend the basic data flow network design [78] [1] [67]. Before discussing contracts, let's review the basics of data flow networks. Their base types are *data objects* and *components* (sometimes called process objects). The components can be *filters*, *sources*, or *sinks*. Filters have an input and an output, both of which are data objects. Sources have only data object outputs and sinks have only data object inputs. A *pipeline* is a collection of components. It has a source (typically a file reader) followed by many filters followed by a sink (typically a rendering algorithm). Pipeline execution, in this context, will be demand driven, meaning that data flow starts with a *pull* operation. This begins at the sink, which generates an *update* request that propagates up the pipeline through the filters, ultimately going to a load balancer (needed to divide the work on the server) and then to the source. The source generates the requested data which becomes input to the first filter. Then *execute* phases propagate down the pipeline. Each component takes the data arriving at its input, performs some operation and creates new data at its output until the sink is reached.

All of the above description is typical of data flow networks. The primary extensions proposed (and implemented inside the VisIt system) that enables large data optimiza-

tions is the *contract*. The *contract* travels up the pipeline along with the pull request (see figure 3.1).

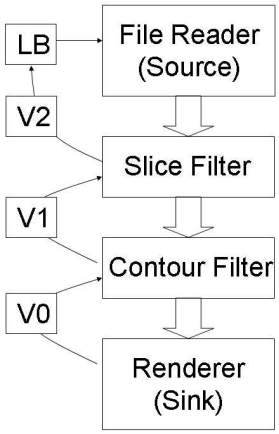

Figure 3.1: An example pipeline. During the update phase (denoted by thin arrows), Contract Version 0 (V0), comes from the sink. V0 is then an input to the contour filter, which modifies the contract to make Contract Version 1 (V1). This continues up the pipeline, until an executive that contains a load balancer (denoted by LB) is reached. This executive decides the details of the execution phase and passes those details to the source, which begins the execute phase (denoted by thick arrows).

The scale of the data sets considered for this dissertation mandates that optimizations are incorporated into every pipeline execution. These optimizations vary from minimizing the data read from disk, to the treatment of that data, to the way that data moves through a pipeline. The set of applicable optimizations is dependent on the properties of the pipeline components. This requires a dynamic system that determines which optimizations can be applied. Further, good software engineering practices mandate consideration for the ease of adding new components. So this system must be able to handle their addition without modification. These requirements lead to our final strategy: all of a pipeline's components modify a contract and optimizations are adaptively employed based on the specifications of this contract.

The heart of the contract-based system is an interface that allows pipeline components to communicate with other filters and describe their *impact* on a pipeline. Focusing on the more abstract notion of *impact* rather than the specifics of individual components allows for a highly extensible architecture, because new components simply must be able to describe what *impacts* they will have. This abstraction also allows for effective management of the large number of existing components.

Because the contract is coupled with update requests, the information in the contract travels from the bottom of the pipeline to the top. When visiting each component in the pipeline, the contract informs that component of the downstream components' re-

quirements, as well as being able to guarantee that the components upstream will receive the current component's requirements. Further, the contract-based system enables all components to participate in the process of adaptively selecting and employing appropriate optimizations. Finally, combining the contract with *update* requests allows for seamless integration into a large system.

3.3 Description of Input Data With Respect to Optimizations

As previously described, the data sets we are considering by VisIt come from parallelized simulation codes. In order to run in parallel, these codes decompose their data into pieces, called *domains*. The domain decomposition is chosen so that the total surface area of the boundaries between domains is minimized, and there is typically one domain for each processor. Again, when these codes write their data to disk, it is typically written in its domain decomposed form. Reading in one domain from these files is usually an atomic operation; the data is laid out such that either it is not possible or it is not advantageous to read in a portion of the domain.

Some data sets provide meta-data, allowing the visualization tool to speed up their processing. Here, meta-data means data about the data set that is much smaller in size than the whole data set. Examples of meta-data are spatial extents for each domain of the simulation or variable extents for each domain for a specific variable of the simulation.

3.4 Optimizations

In the following sections, some of example optimizations will be described. The potential application of these optimizations is dependent on the properties of a pipeline's components. A contract-based system is necessary to facilitate these optimizations being applied adaptively.

After the optimizations are described, the contract-based system will be presented in more detail.

3.4.1 Reading the Optimal Subset of Data

I/O is the most expensive portion of a pipeline execution for almost every operation a visualization tool performs. But the amount of time spent in I/O can be minimized by reading only the domains that are relevant to any given pipeline. This performance gain propagates through the pipeline, since the domains not read in do not have to be processed downstream.

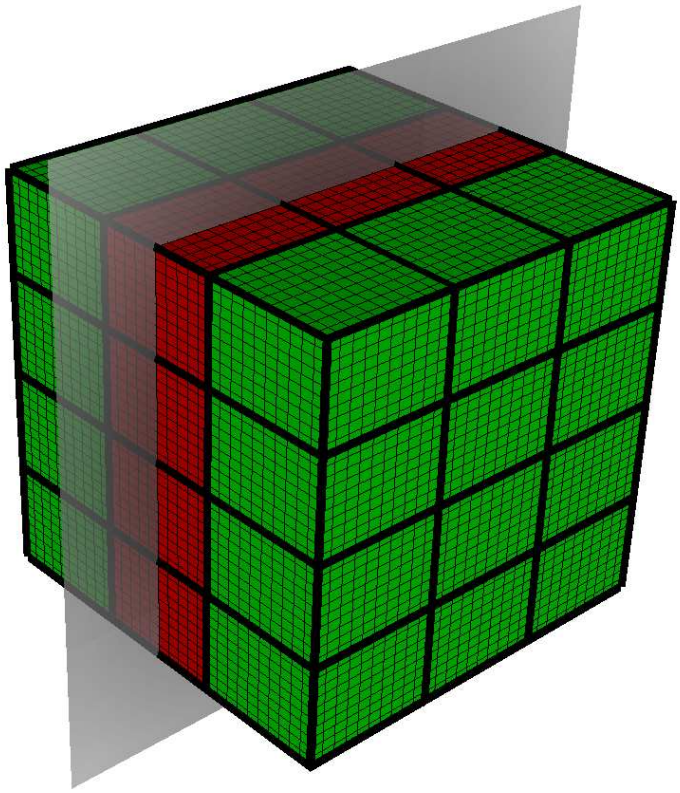

Figure 3.2: Shown is a 36 domain data set. The domains have thick black lines and are colored red or green. Mesh lines for the elements are also shown. To create the data set sliced by the transparent grey plane, only the red domains need to be processed. The green domains can be eliminated before ever being read in.

Consider the example of slicing a three-dimensional data set by a plane (see figure 3.2). Many of the domains will not intersect the plane and reading them in will be wasted effort. In fact, if D is the total number of domains, then the number of domains intersected by the slice is typically $O(D^{2/3})$.

With the presence of meta-data, it is possible to eliminate domains from processing before ever reading them. For example, if the slice filter had access to the spatial extents

for each domain, it could calculate the list of domains whose bounding boxes intersects the slice and only process that list (note that false positives can potentially be generated by considering only the bounding box).

The contract methodology enables this. During the update phase, every filter is given an opportunity to modify the contract, which contains the list of domains to be processed. A filter can check to see if some piece of meta-data is available (for example, spatial extents), and, if so, cross-reference the list of domains to be processed with the meta-data. The modified contract will then contain only those domains indicated by the filter.

It is important to note that every filter in the pipeline has a chance to modify the contract. If a pipeline had a slice filter and a contour filter (to generate isolines), the slice filter could use a spatial extents meta-data object to get exactly the set of domains that intersected the slice, while the contour filter could use a data extents meta-data object to get exactly the set of domains that could possibly produce contours. The resulting contract would contain the intersection of their two domain lists.

Further, since the infrastructure for subsetting the data is encapsulated in the contract, new, unforeseen filters can leverage this optimization. For example, a newly added spherical-slice filter can be added afterwards and it can also make use of the spatial extents meta-data, or a new filter that thresholds the data to produce only the elements that meet a certain criteria (elements with density between 2 g/cc and 5 g/cc, for example) can use the data extents meta-data. Also, the types of meta-data that fit within a contract scheme are not limited to spatial and data extents. There is an abstract meta-data type, and new, concrete types of meta-data can take any form.

3.4.2 Execution Model

For domain-decomposed data sets, our strategy is to leverage the domain decomposition of the simulation for our own parallelization model. The number of processors the parallel server is run on is typically much less than the number of domains the simulation code produced. So we must support *domain overloading*, where multiple domains are processed on each processor of the server. Note that it is generally not sufficient to

simply combine unrelated domains into one larger domain. This is not even possible for some grid types, like rectilinear grids where two grids are likely not neighboring and cannot be combined. And for situations where grids can be combined, as with unstructured grids, additional overhead would be incurred to distinguish which domains the elements in the combined grid originated from, which is important for operations where users want to refer to their elements in their original form (for example, picking elements).

We have considered two techniques to do domain overloading. One approach, called *streaming*, processes domains one at a time. In this approach, there is one pipeline execution for each domain. Another approach, called *grouping*, executes the pipeline only once and has each component process all of the domains before proceeding to the next one.

A related issue to domain overloading is that of load balancing. With *static load balancing*, domains are assigned to the processors at the beginning of the pipeline execution and a grouping strategy is applied. Because all of the data is available at every stage of the pipeline, collective communication can take place, enabling algorithms that cannot be efficiently implemented in an out-of-core setting. With *dynamic load balancing*, domains are assigned dynamically and a streaming strategy is applied. Not all domains take the same amount of time to process; dynamic load balancing efficiently (and dynamically) schedules these domains, creating an evenly distributed load. In addition, this strategy will process one domain in entirety before moving on to the next one, increasing cache coherency. However, because the data streams through the pipeline, it is not all available at one time. This makes some parallel algorithms which require all of the data to be in memory at one time impossible to implement. Algorithms of this form typically utilize collective communication, where data from one chunk is needed to perform calculations on another chunk, possibly many, many times, like with parallel streamline generation.

So which load balancing method should be used? Dynamic load balancing is more efficient when the amount of work per domain varies greatly, but the technique does not support all algorithms. Static load balancing is usually less efficient, but does support all algorithms. The best solution is to use dynamic load balancing when possible, but fall back on static load balancing when an algorithm can not be implemented in a streaming setting. A contract-based system is again used to solve this problem. When each pipeline

component gets the opportunity to modify the contract, it can specify whether or not it will use collective communication. When the load balancer executes, it will consult the contract and then use that information to adaptively choose between dynamic and static load balancing.

Note that the load balancing described in this section refers to the processing of data. This processing can be varied, for example generating a surface to be rendered, or subselecting a region and then performing a query on it, such as integrating some quantity. In the case of generating a surface to be rendered, there is additional load balancing that will improve rendering speed. When the actual rendering occurs, it is often lucrative to perform a re-balancing of the surface so that each processor contains approximately the same number of primitives. This form of load balancing is distinct from what is considered in this section.

3.4.3 Generation of Ghost Data

Up until this point, we have described an approach where data flow networks operate on chunks of a larger data set. On the whole, this is an excellent (and scalable) approach. However, we have not considered the problems that can arise from that. The biggest problem occurs along the exterior layer of elements for each chunk. In the following paragraphs, we will describe different types of artifacts that can occur and how they can be overcome.

One common operation is to remove a portion of a data set (for example, clipping a wedge out of a sphere) and then look at only the external faces of what remains. This can be done by finding the external faces of each of the data set's domains. But faces that are external to a domain can be internal to the whole data set. These extra faces can have multiple negative impacts. One impact is that the number of triangles being drawn can go up by an order of magnitude. Another impact occurs when the external faces are rendered transparently. Then the extra faces are visible and result in an incorrect image (see figure 3.3).

Now consider the case where interpolation is needed to perform a visualization operation. For example, consider the case where a contour is to be calculated on a data

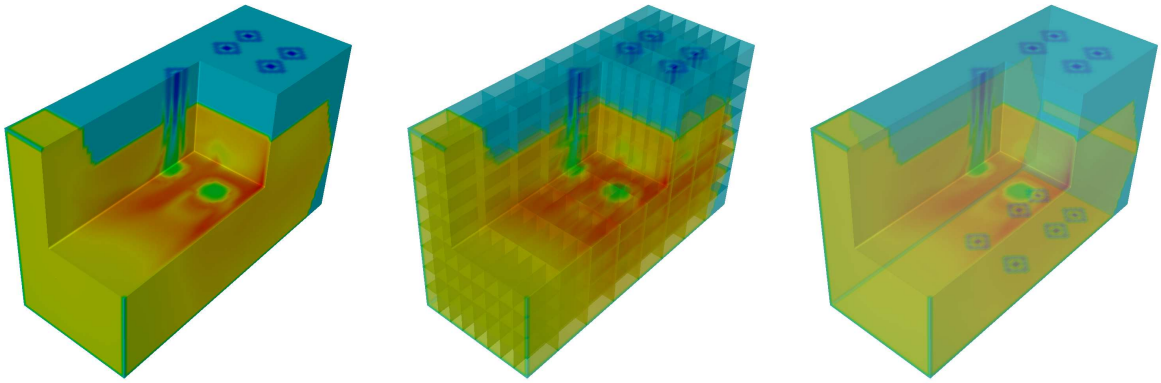

Figure 3.3: On the left is an opaque picture of the data set. In the middle, the opacity has been lowered. Faces external to a domain (yet internal to the data set) are being rendered. On the right, these faces have been removed. There are 902,134 triangles for the middle surface and only 277,796 for the right surface.

set that has an element-centered scalar quantity defined on it. Since contouring is typically done with an algorithm that requires node-centered data, the first step of this process is to interpolate the data to be a node-centered quantity from an element-centered quantity. Along the domain boundaries, the interpolation will be incorrect, because the elements from neighboring domains are not available. Ultimately, this will lead to a cracked contour surface (see figure 3.4).

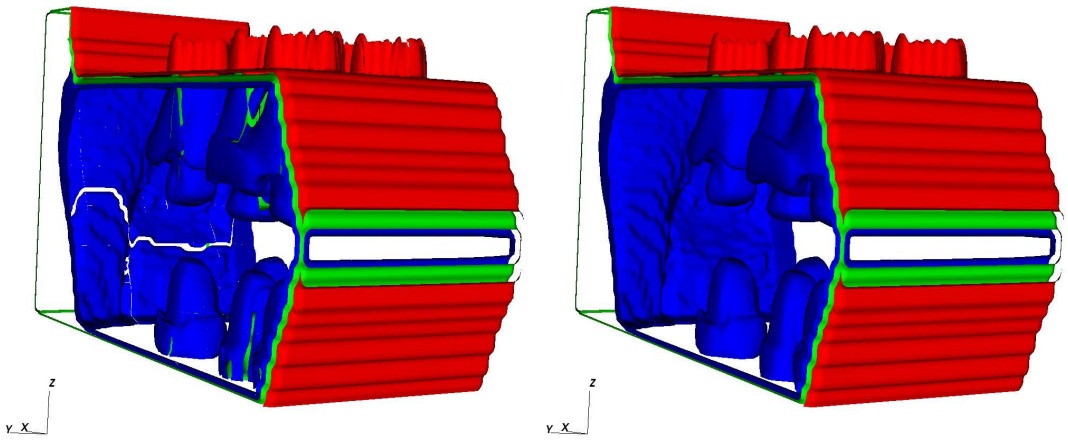

Figure 3.4: On the left is a contour plot of an element-centered quantity where ghost elements were not generated. Cracks in the contour surface occur along domain boundaries. On the right, ghost elements were generated and the correct picture was generated.

Both of the above problems require ghost data. For the first case, it is sufficient to mark the exterior faces of a domain that are internal to the whole data set as ghost faces.

Then these ghost faces can be discarded when an external face list is generated. The second case requires a redundant layer of ghost elements around the exterior of each domain. This allows interpolation to be done correctly.

Generation of ghost data is typically possible given some description of the input data set. This input can take several forms. One form can be a description of how the domain boundaries of structured meshes overlap ("faces I=0-5, J=0, K=8-12 of domain 5 are the same as faces I=12-17, J=17, K=10-14 of domain 12"). Another form utilizes global node identifiers assigned by the simulation code for each node in the problem. The visualization tool can then use these identifiers to determine which nodes are duplicated on multiple domains and thus identify shared boundaries between domains, which is the key step for creating ghost data. A third form uses the spatial coordinates of each node as a surrogate for global node identifiers. For each of these forms, the salient issue is that a module can be written that can create ghost faces or ghost elements. The details of such a module are not important to this dissertation.

There are costs associated with ghost data. Routines to generate ghost elements are typically implemented with collective communication, which precludes dynamic load balancing. In addition, ghost elements require duplicate copies of some cells in the data set (see figure 3.5), increasing memory costs. Ghost faces are less costly, but still require arrays of problem size data to track which faces are ghost and which are not. To this end, it is important to determine the exact type of ghost data needed, if any.

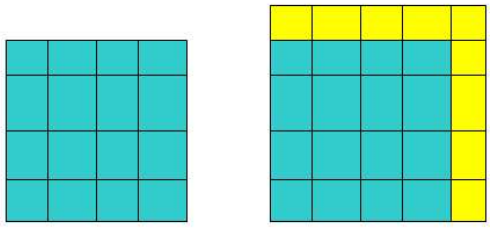

Figure 3.5: On the left is a domain without ghost elements. On the right is the same domain with ghost elements added (drawn in yellow). VisIt combines the ghost elements with the domain's real elements into one large domain for efficiency purposes for the filters downstream as well as simplicity of coding.

A contract-based system allows the minimum calculation to be performed. If a

filter, such as the contour filter, needs to do interpolation, it will mark the contract with this information in the update phase. As a result, ghost elements will be created at the top of the pipeline, allowing correct interpolation to occur. If the filter believes it will have to calculate external face lists, the case with the external face list filter, then it will mark the contract with this information in the update phase, and ghost faces will be created. Most importantly, in cases that do not require ghost data, such as the case when a data set is being sliced or volume rendered no ghost data will be created[2].

3.4.4 Subgrid Generation

Before discussing subgrid generation, first consider "clip" and "threshold" filters. Clipping allows a user to remove portions of a data set based on standard geometric primitives, such as planes or spheres. Thresholding allows the user to generate a data set where every element meets a certain criteria – the elements where density is greater than 2 g/cc and the temperature is between six hundred and eight hundred degrees Celsius. Both of these filters produce unstructured grid outputs even if the input grid is structured.

Our experience has been that most simulations with the largest number of elements are performed on rectilinear grids. Rectilinear grids have an implicit representation that minimizes the memory footprint of a data set. Many filters, such as clip and threshold, take in rectilinear grid inputs and create unstructured grid outputs. One issue with the unstructured grid outputs is that many components have optimized routines for dealing with rectilinear grids. A bigger issue is that of memory footprint. The representation of an unstructured grid is explicit, and the additional memory required to store them can be more than what is available on the machine.

To further motivate this problem, consider the following example of a ten billion element rectilinear grid with a scalar, floating-point precision variable. The variable will occupy forty gigabytes (40GB) of memory. The representation of the grid itself takes only a few thousand bytes. But representing the same data set as an unstructured grid is much more costly. Again, the scalar variable will take forty gigabytes. Each element of the grid

[2]Ghost data is clearly not required for cell-centered data in these cases. For node-centered data, the data at the boundaries of a chunk is often repeated and additional ghost data is not necessary in this case as well.

will now take eight integers to store the indices of the element's points in a point list, and each point in the point list will now take three floating-point precision numbers, leading the total memory footprint to be approximately four hundred eighty gigabytes (480GB). Of course, some operations dramatically reduce the total element count – a threshold filter applied to a ten billion element rectilinear grid may result in an unstructured grid consisting of just a few elements. In this case, the storage cost for an unstructured grid representation of the filter's output is insignificant when compared to the storage cost of the filter's input. However, the opposite can also happen: a threshold filter might remove only a few elements, creating an unstructured grid that is too large to store in memory.

One way to address this problem is to identify complete rectilinear grids in the filter's output. These grids are then separated from the remainder of the output and remain as rectilinear grids. Accompanying this grid is one unstructured grid that contains all of the elements that could not be placed in the output rectilinear grids. Of course, proper ghost data is put in place to prevent artificial boundary artifacts from appearing (which type of ghost data is created is determined in the same way as described in 3.4.3). This process is referred to as *subgrid generation* (see figure 3.6).

We rejected the alternative approach of keeping the grid in its complete form and adding a bit to each cell stating wither or not it survived. This approach does not lend itself to algorithms where cells are divided into tetrahedra (such as clipping). Worse, the approach would create new API requirements on the other filters to be aware of this new bit array and incorporate it into their processing.

There are many different configurations where rectilinear grids can be overlaid onto the "surviving elements" in the unstructured grid. The best configuration would maximize the number of elements covered by the rectilinear grids and minimize the total number of rectilinear grids. These are often opposing goals. Each element could be covered by simply devoting its own rectilinear grid to it. Since each grid has overhead, that would actually have a higher memory footprint than storing them in the original unstructured grid, defeating the purpose.

Although our two goals are opposing, we are typically more interested in one goal than another. For example, if we are trying to volume render the data set, then the

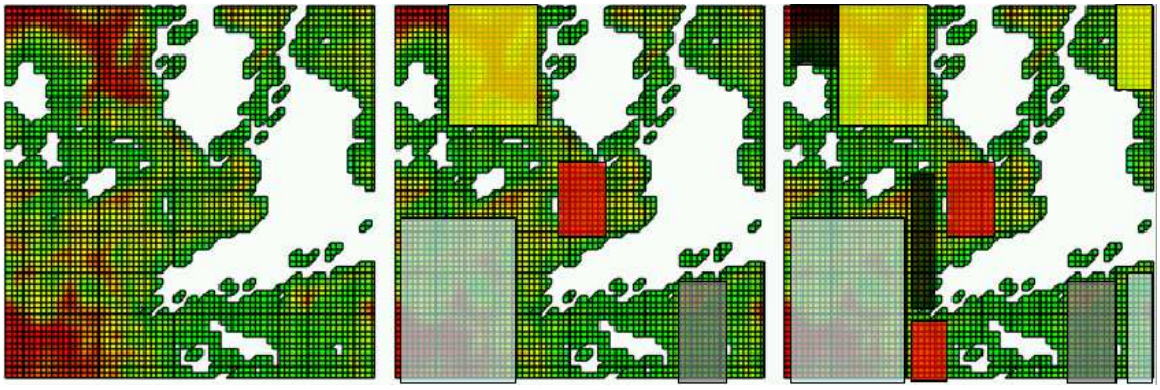

Figure 3.6: On the left, there is rectilinear grid with portions removed. In the middle, we see a covering with a large minimum grid size, which results in four grids. On the right, we see a covering with a smaller minimum grid size, which results in nine grids. The elements not covered in the output grids are placed in an unstructured grid.

performance of the algorithm is far superior on rectilinear grids than on unstructured grids. In this case, we would want to make sure the maximum number of elements was covered by the rectilinear grids. But the performance of many operations is indifferent to grid type, making memory footprint the only advantage for those operations.

A contract-based system can guide the placement of the rectilinear grids. Each component can modify the contract to say which goal it values - solely minimizing memory footprint versus keeping as many elements as possible in a native rectilinear representation for further processing.

As previously mentioned, some complete rectilinear grids contain so few elements that leaving them in an implicit form does not provide a favorable memory tradeoff, because there is overhead associated with each rectilinear grid. As such, we use a minimum grid size of 2048 elements when overlaying complete rectilinear grids on the output. However, if the contract reports that filters down stream can take advantage of rectilinear representations, then the minimum grid size drops to 256 elements. Performance results for differing grid sizes can be found in 3.6.4.

3.5 Description of Contract

The contract is simply a data structure. An initial version is created at the sink with all default values. As each component of the pipeline is visited during the update

phase, it can modify the contract by changing members of this data structure. Table 3.1 contains a description of the members of the contract referenced in the previous sections.

Name	Type	Default Value
ghostType	enum {None Face, Element}	None
optimizedFor-Rectilinear	bool	false
canDoDynamic	bool	true
domains	vector<bool>	all true

Table 3.1: The members of the contract data structure described in previous sections.

For our implementation in VisIt, each filter inherits from a base class, called *avtFilter*. This class has a virtual function that allows the filter to modify the contract. Below is pseudocode for how to modify a contract.

```
void avtXYZFilter::ModifyContact(avtContract *c)
{
        c->SetCanDoDynamic(false);
        c->SetGhostType(Element);
}
```

The contract allows for each component to describe the impact it will have on the total pipeline execution in a general way that does not require knowledge of the component itself. By enumerating the impacts that components can have on a pipeline, it is easy to extend this system with new components. In addition, by using inheritance, the burden to implement a new component and utilize the contract-based system is very low.

The *ghostType* data member is set to *None* by default, because ghost data should not be generated when it is not required. The contour filter modifies the contract to have *Element* when it is going to contour element-based data, whereas the external face list filter modifies the contract to have *Face*. It should be noted that all of the components that modify this field do not blindly assign their desired value to it. If the face list filter were to

overwrite a value of *Element* with its desired *Face*, then the filters downstream would not get the ghost data it needs. In this case, there is an ordering between the types. If *Element* was requested, then it should be obtained, regardless of requests for *Face* data. Similarly, *Face* data should be obtained even if other filters need *None*. And those that request *None* should gracefully accept and pass through ghost data. Furthermore, those that request *Face* data should be able to accept and deal gracefully with *Element* data in its place. The external face list filter, then, is able to accommodate *Element* data, even when it simply needs *Face* data. Although it would be possible to eliminate this complexity (by having separate entries in the contract for ghost faces and ghost elements), the field is maintained as one entry because it is more efficient to only calculate one set of ghost data.

optimizedForRectilinear is set to false by default, since only certain filters are specialized for operating on rectilinear grids. If the field is false, then the grids are placed to minimize memory footprint, rather than maximizing the number of elements covered by the rectilinear grids. *canDoDynamic* is set to true because it assumed that most filters do not require collective communication. If they do require collective communication, it is their responsibility to set that *canDoDynamic* to false when it has a chance to modify the contract in the update phase. Finally all of the *domains* are assumed to be used at the beginning of the update. If filters are able to access meta-data and determine that some domains will not affect the final picture, then they may modify the Boolean values for those domains.

3.6 Results

The contract-based system described in this chapter has been fully implemented and we now discuss its results. The purpose of discussing the results is to demonstrate the benefit of the example optimizations discussed. We believe that this motivates the importance of using these optimizations and, by extension, motivates the importance of a contract-based system that enables these specialized optimizations to be adaptively employed.

We will present results in the context of a Rayleigh-Taylor Instability simulation,

which models fluid instability between heavy fluid and light fluid. The simulation was performed on a 1152x1152x1152 rectilinear grid, for a total of more than one and a half billion elements. The data was decomposed into 729 domains, with each domain containing more than two million elements.

All timings were taken on Lawrence Livermore National Laboratory's *Thunder* machine, which was ranked seventh on the Top 500 list released in June 2005. The machine is comprised of 4096 1.4GHz Intel Itanium2 processors, each with access to two gigabytes of memory. The machine is divided into 1024 nodes, where each node contains four processors. The processor's memory can only be shared with other processors in its node.

Pictures of the operations described below are located at the end of this chapter.

3.6.1 Reading the Optimal Subset of Data

We will present two algorithms where the optimal subset of data was read – slicing, which makes use of spatial meta-data, and contouring, which makes use of variable meta-data. It should be noted that use of spatial meta-data typically yields a consistent performance improvement, but performance improvement from variable meta-data can be highly problem specific. To illustrate this, results from early in the simulation and late in the simulation are shown (see table 3.2). The processing time includes the time to read in a data set from disk, perform operations to it, and prepare it for rendering. Rendering was not included because it can be highly dependent on screen size.

		Processing time (sec)	
Algorithm	Processors	Without Meta-data	With Meta-data
Slicing	32	25.3	3.2
Contouring (early)	32	41.1	5.8
Contouring (late)	32	185.0	97.2

Table 3.2: Measuring effectiveness of reading the optimal subset of data

3.6.2 Comparison of Execution Models

Since not all pipelines can successfully execute with dynamic load balancing, we can only compare execution time for those pipelines that can use dynamic load balancing. Again using the Rayleigh-Taylor Instability simulation, we study the performance of slicing, contouring, thresholding, and clipping. Note that other optimizations were used in this study – slicing and contouring were using spatial and variable meta-data respectively, while thresholding and clipping used subgrid generation for its outputs (see table 3.3). Again, the processing time includes the time to read in a data set from disk, perform operations to it, and prepare it for rendering.

		Processing time (sec)	
Algorithm	Processors	Static LB	Dynamic LB
Slicing	32	3.2	4.0
Contouring	32	97.2	65.1
Thresholding	64	181.3	64.1
Clipping	64	59.0	30.7

Table 3.3: Measuring performance differences between static and dynamic load balancing

Slicing did not receive large performance improvements from dynamic load balancing, because our use of spatial meta-data eliminated those domains not intersecting the slice, and the amount of work performed per processor was relatively even. We believe that the higher dynamic load balancing time is due to the overhead in multiple pipeline executions. Contouring, thresholding, and clipping, on the other hand, did receive substantial speedups, since the time to execute each of these algorithms was highly dependent on its input domains.

3.6.3 Generation of Ghost Data

This optimization is not a performance optimization; it is necessary to create the correct picture. Hence, no performance comparisons are presented here. Refer back to figures 3.3 and 3.4 in Section 3.4.3 to see the results.

3.6.4 Subgrid Generation

The volume renderer for this study will be described in more detail in chapter 4. We will now give a brief description of the algorithm to understand how contracts benefit such an algorithm. The volume renderer processes data in three phases. The first phase samples the data along rays. The input data can be heterogeneous, made up of both rectilinear and unstructured grids. The rectilinear grids are sampled quickly using specialized algorithms, while the unstructured grids will be sampled slowly using generalized algorithms. The sampling done on each processor uses the data assigned to that processor by the load balancer. Once the sampling has been completed, the second phase, a communication phase, begins. During this phase, samples are re-distributed among processors, to prepare for the third phase, a compositing phase. The compositing is done on a per-pixel basis. Each processor is responsible for compositing some portion of the screen, and the second communication phase brings the samples necessary to perform this operation.

The volume renderer uses the contract to indicate that it has rectilinear optimizations. This will cause the subgrid generation module to create more rectilinear grids, many of them smaller in size than what is typically generated. This then allows the sampling phase to use the specialized, efficient algorithms and finish much more quickly.

In the results below, we list the time to create one volume rendered image (see figures 3.12 and 3.13). Before volume rendering, we have clipped the data set or thresholded the data set and used subgrid generation to create the output. Table 3.4 measures the effectiveness of allowing for control of the minimum grid size (2048 versus 256) with subgrid generation. When subgrid generation was not used, only unstructured grids were created, and these algorithms exhausted available memory, leading to failure with this number of processors.

It should be noted that the rendering time is dominated by sampling the unstructured grids. This data set can be volume rendered in 0.25 seconds when no operations (such as clipping or thresholding) are applied to it.

The thresholded volume rendering produces only marginal gains, since the fluids have become so mixed that even rectilinear grids as small as 256 elements cannot be placed

		Subgrid Generation		
		No	Yes	
			Minimum Grid Size	
Algorithm	Processors		2048	256
Clip	64	Out Of Memory	12.0s	9.0s
Thresholding	64	Out Of Memory	11.4s	10.8s

Table 3.4: Measuring effectiveness of grid size control with subgrid generation

over much of the mixing region.

3.7 Summary

The scale of data considered for this dissertation requires that as many optimizations as possible be included in each pipeline execution. Yet, in the real world, large number of components, including the addition of new plugin components, makes it difficult to determine which optimizations can be applied. A contract-based system solves this problem, allowing all possible optimizations to be applied to each pipeline execution. The contract is a data structure that extends the standard data flow network design. It provides a prescribed interface that every pipeline component can modify. Furthermore, the system is extensible, allowing for further optimizations to be added and supported by the contract system.

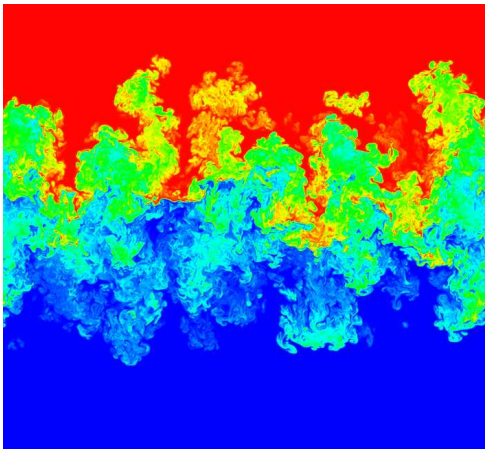

Figure 3.7: This is a slice of the simulation at late time. Light fluid is colored blue, heavy fluid is colored red.

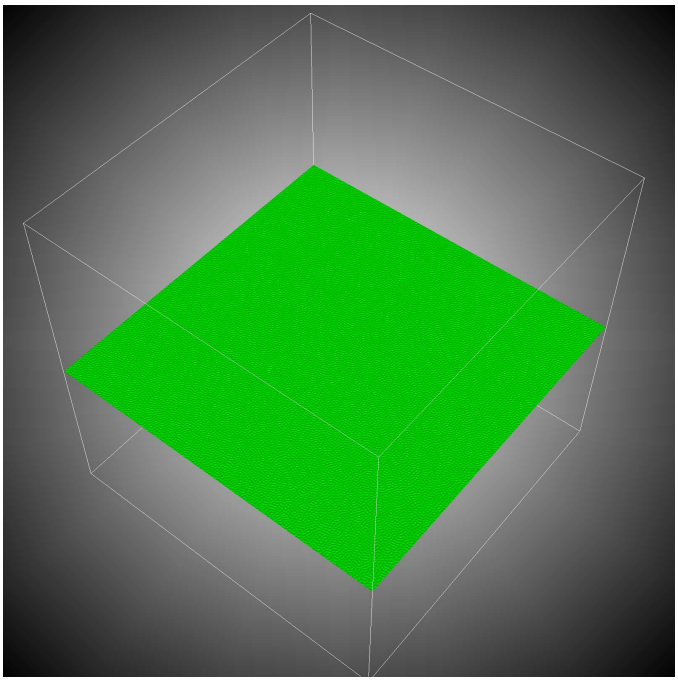

Figure 3.8: A contour at early simulation time. This contour separates the light and dense fluids.

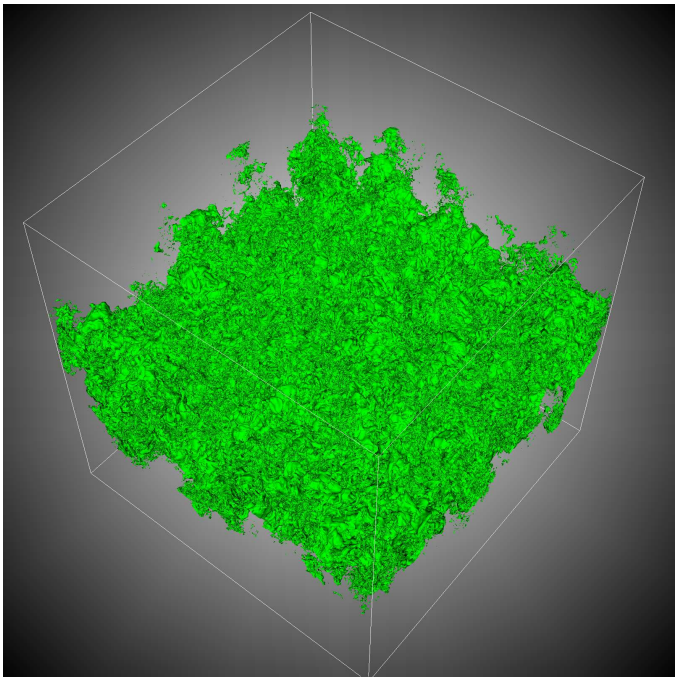

Figure 3.9: A contour at late simulation time. This contour separates the light and dense fluids.

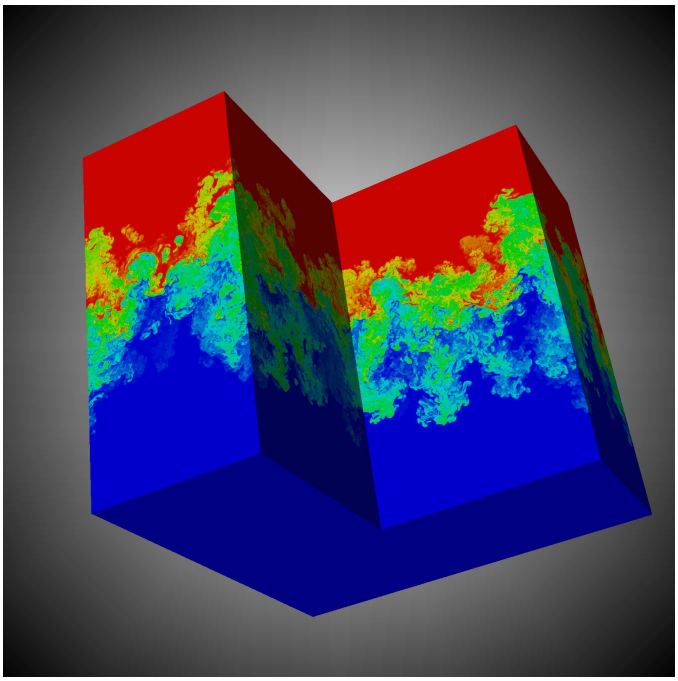

Figure 3.10: Pictured here is the simulation with one portion clipped away. Light fluid is colored blue, heavy fluid is colored red.

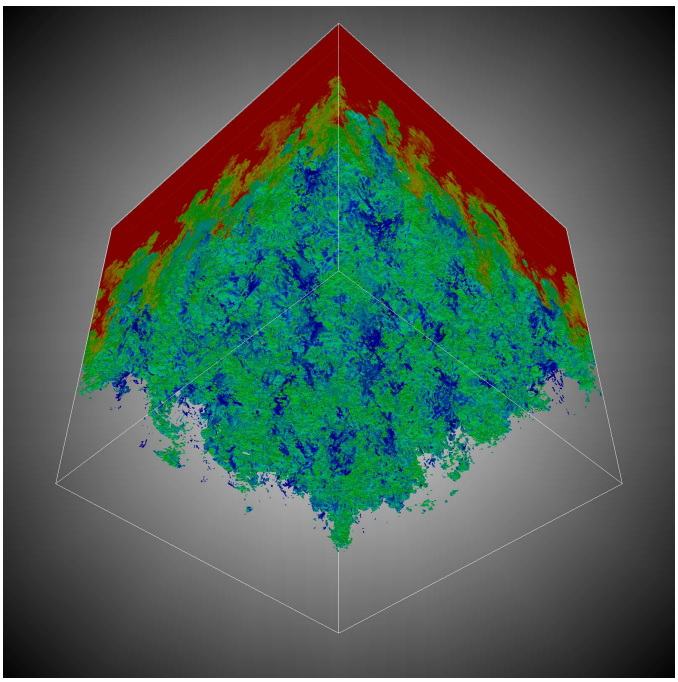

Figure 3.11: Pictured here is the simulation with the light fluid removed using the threshold operation.

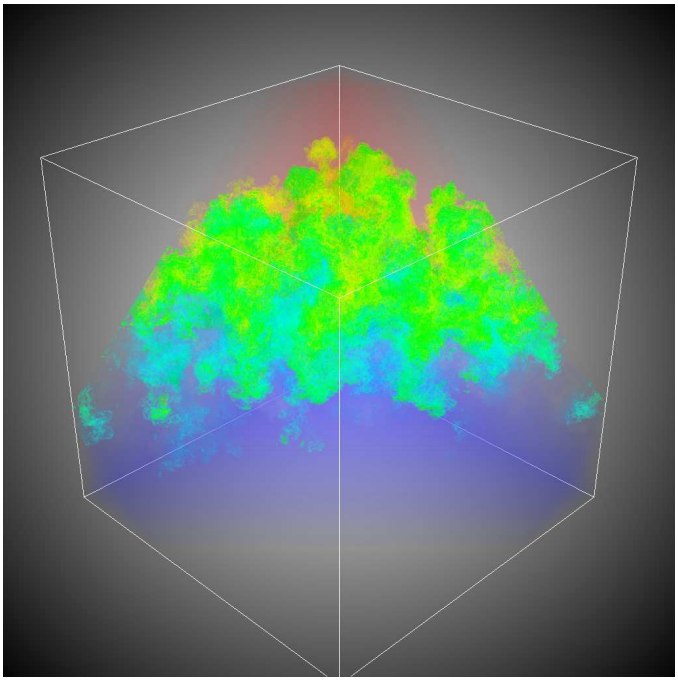

Figure 3.12: Pictured here is a volume rendering of the simulation after being clipped by a plane.

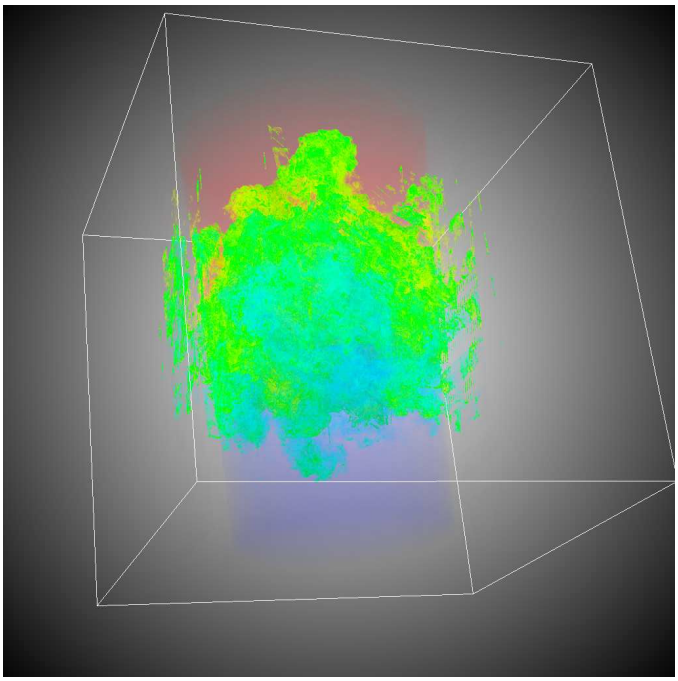

Figure 3.13: This is a volume rendering of the simulation after thresholding by a distance variable to remove elements outside a cylinder.

Chapter 4

A Scalable, Hybrid Scheme for Volume Rendering Massive Data Sets

The architecture presented in the previous two chapters is well suited for algorithms where the data can be processed in sub-pieces and the results of that processing can be trivially combined. Examples of such algorithms include contouring and slicing. Volume rendering, on the other hand, when implemented in a parallel setting, requires great care to be taken when dividing work among processors and thus achieve good parallel efficiency.

In this chapter[1], we describe a parallel, distributed memory algorithm for volume rendering massive data sets. The algorithm's scalability has been demonstrated up to 400 processors, rendering one hundred million unstructured elements in under one second. The heart of the algorithm is a hybrid approach that parallelizes over both the elements of the input data and over the pixels of the output image. At each stage of the algorithm, there are strong limits on how much work each processor performs, ensuring good parallel efficiency. The algorithm is sample-based. We present two techniques for calculating the sample points: a 3D rasterization technique and a kernel-based technique, which trade off between speed and generality. Finally, the algorithm is very flexible. It can be deployed in

[1]Much of the text from this chapter comes from [18], which was a collaboration between Hank Childs, Mark Duchaineau, and Kwan-Liu Ma

general purpose visualization tools and can also support diverse mesh types, ranging from structured grids to curvilinear and unstructured meshes to point clouds.

4.1 Introduction

The power and capacity of parallel supercomputers are growing at a fast pace, enhancing scientists' ability to simulate and study complex physical phenomena at increasingly greater accuracy. This added accuracy often is the key to advances in their studies. Their 3D simulations can generate large volume data sets, which are best visualized using direct volume rendering. Volume rendering is particularly effective in displaying complex 3D structures and features at varying scales. Real-time volume rendering has become feasible with commodity graphics hardware, but it requires the data to fit in video memory. Because the size of the data sets generated by these simulations are currently in the terascale regime, and will soon reach petascale, we must seek a distributed-memory, parallel rendering solution.

We have developed a new, massively parallel, distributed-memory, volume rendering algorithm. It is designed to run on the same parallel supercomputers used to run terascale/petascale simulations. It has demonstrated excellent scalability and been used on some of the largest simulations ever performed. The algorithm can be applied to many diverse types of meshes, ranging from structured grids to curvilinear and unstructured meshes to point clouds. It is implemented inside of VisIt [17], a richly featured visualization and data analysis application focusing on large data sets. Throughout the development of this algorithm, a major goal was to develop a technique that would seamlessly integrate into a general-purpose visualization tool. This required us to avoid schemes that place special requirements on I/O, communication, available hardware, etc.

In a distributed memory environment, adaptive techniques for dividing work among the processors are not possible due to data ownership issues. In this environment, data must be partitioned among the processors and each processor is responsible for calculating its portion of the picture. Of course, load balancing issues can occur, especially with irregular data or when camera placement focuses on a small sub-region of the volume. This requires

us to develop data partitioning strategies that ensure even workloads on each processor. The quality of these strategies dictate the degree of scalability we can achieve at high processor counts.

We employ a multi-phase approach for distributing work. This technique is a hybrid of previous techniques and allows us to incorporate their best aspects with respect to balanced processing. During each phase of our algorithm, our data organization naturally enforces hard limits on how much work each processor can do. This creates excellent load balancing which in turn creates excellent scalability.

In this chapter, we present the details of our algorithm, a scaling study, and some applications. Our algorithm is sample-based, which requires us to implement custom sampling techniques. We present a 3D rasterization technique and a kernel-based technique, which trade off between speed and quality, respectively. The scaling study shows linear scalability of our algorithm up to four hundred processors. In addition, we present the performance of this algorithm for diverse mesh types: a one billion-particle point cloud simulation, a 27 billion-element rectilinear mesh simulation, and a one hundred-forty million element unstructured mesh simulation.

4.2 Related Work

Among the parallel rendering algorithms specifically designed for visualizing irregular-grid volume data, the most relevant example for this work is the sort-last cell-projection algorithm introduced by Ma and Crockett [44]. This distributed-memory algorithm achieves high performance by completely overlapping rendering and compositing, and by keeping sorting costs to a minimum. However, its high scalability partially relies on manual setting of communication parameters. The other class of algorithms, those that require a view-dependent sorting step, are not feasible when rendering terascale and petascale data sets.

Wang, Gao and Shen [80] use a multi-resolution wavelet tree to allow parallel volume rendering of large data in an error-guided fashion. In contrast, our goal is to render the data at its highest possible resolution.

There is a growing interest in parallel GPU-based volume rendering [41, 75, 15]. Nevertheless, a PC cluster that is capable of rendering irregular-grid data sets at the scale we are faced would be prohibitively expensive to build. Our algorithm is designed to harness the power of parallel supercomputers. Another option is to utilize geographically distributed computing resources, which, if appropriately coordinated, could become very attractive. Gao et al. demonstrated such an approach with parallel rendering of large volume data distributed over a wide area network [28].

4.3 Algorithm Overview

The algorithm itself consists of two high-level phases. In the first phase, samples are generated from the input data set. In the second phase, those samples are classified and composited to form the final picture.

Our goal in the sampling phase, for a given view frustum, is to determine the value of billions of samples along millions of rays, where there is one ray for each pixel of the image and thousands of samples along each ray. In abstract terms, a sampling algorithm takes a data set as input, as well as parameters describing the volume rendering requirements: the view frustum (i.e. camera location, view direction, view angle, and near and far clipping planes), the screen size (number of pixels in width, W, and height, H), and the number of samples, S, to take along each ray.

Given these inputs, the goal of a sampling algorithm is to create an output with the following properties:

1. A logically structured grid, G, of dimensions WxHxS
2. A field F defined on G, which has sampled the values of the input data set
3. Each F[w, h, s] corresponds to the value of s^{th} sample along the ray corresponding to pixel (w, h) of the output image for that view frustum

Given a black box that performs a sampling algorithm, it is trivial to implement an algorithm for the second, compositing phase. For each pixel (w, h) and each sample for that pixel, we can classify the field value based on some user defined transfer function. Then the

resulting colors and opacities can be composited using the "under" function (front-to-back compositing) suggested by Porter and Duff[62] to create the color for that pixel. Doing this for all pixels yields the final, volume rendered picture.

We also note that a sample-based approach makes for easy integration with renderings of standard geometric primitives. An additional image input to the compositing module can specify background colors and prematurely terminate rays when they reach the depth of the first encountered geometric primitive for that pixel. This image is generated on a previous render pass.

By using samples as our fundamental unit, we have made sacrifices. We lose the ability to resolve complex interactions between elements, like the type addressed in [22]. But, because the samples are calculated on a per view frustum-basis, this issue is greatly mitigated. It is our belief that, with a sufficient number of samples per ray and by taking care in assigning the individual sample values, a sample-based method can provide good results. This approach of choosing a representative value for a sample is reminiscent of the clustering approach presented in [33], although at a different resolution.

4.4 Hybrid Sampling Algorithm

In the following subsections, we give an overview of the data management of the hybrid sampling scheme, without consideration of how to infer the values of the sample points. These subsections address how to organize the data in an efficient way for parallelization and how to store the field F without exceeding primary memory. The purpose of our technique is to address the load balancing issues in a distributed memory environment. In a shared memory environment, such as the one described in [25], many of these load balancing issues do not occur because the parallelization across image space can be done adaptively.

The principal contribution of our data organization scheme is obtaining good load balance for meshes with great variation in the spatial density of their elements. For these meshes, in the extreme, the number of elements within a region of space can differ by orders of magnitude. Schemes that parallelize over the output image's pixels suffer extreme load

imbalance with these meshes, because some pixels will cover many more elements within their projection than others. Similarly, schemes that parallelize over the input data set's elements suffer extreme load imbalance, because some elements will take up disproportionately large portions of the view frustum.

Our multi-stage algorithm, presented in the following subsections, is well balanced in the amount of elements *and* the portion of the view frustum processed. Our hybrid scheme utilizes the best portions of schemes that parallelize over the output's image and schemes that parallelize over the input's elements, while avoiding their pitfalls. It does this by processing the data in stages. In the first stage, it parallelizes across the elements of the input data set, but defers processing of the large elements. In a subsequent stage, it parallelizes across the output image and only then processes the large elements. Again, this organization of the data processing places hard limits on the work performed at each stage, ensuring good parallel efficiency.

If a data set has large spatial variation when projected into screen space, it can greatly complicate load balancing. Again, one reason our algorithm is noteworthy is because it is well suited to handling these data sets. There are two ways that these spatial variations can arise. First, it occurs consistently for meshes that have great variation in spatial density in world space. Second, it can occur to meshes of relatively uniform density in world space through the camera transformation. An example is when the camera is placed in the middle of the data set, which often happens during fly-throughs in movies. In this case, even if the input mesh has uniform-sized elements, elements that are near the camera will occupy orders of magnitude larger portions of the view frustum than those farther from the camera.

Our sampling algorithm contains a total of three stages, characterized as Small-Element Sampling, Communication, and Large-Element Sampling. Also, the algorithm is a dual partition scheme; one partition is of the input data, the other partition is of the rays. In the Small-Element Sampling stage, the first partition is used. The Communication stage re-distributes between the two partitions. And the Large-Element Sampling stage uses the second partition.

Section 4.4.1 discusses the two partitions, 4.4.2 discusses the three phases of the algorithms, and 4.4.3 discusses the load balancing.

4.4.1 Partitions

The partition of the input data is done by the greater VisIt system and is guaranteed to assign approximately equal numbers of elements to each processor. Although we might be able to modestly reduce communicate costs by re-partitioning the data, we do not employ this technique. First, given the massive size of the data sets we are operating on, it is not viable to re-partition all of the data for each rendering. The only viable re-partitioning scheme would be to re-partition one time and use that for all subsequent renderings. However, even if we did perform a one-time re-partition, it is difficult to find one that will truly help with parallel efficiency. It is often not possible to create partitionings that are balanced in both number of elements and spatial footprint. Further, this will not mitigate the issue when the camera is in the middle of the data set.

The second partition is of the samples. We start by dividing the image's pixels among the processors. The division of the samples then follows naturally. Our goal with this partition is to re-assign the samples so that each processor can composite its pixels with no further communication.

4.4.2 Sampling

We define "small" elements as elements that cover a small number of samples, for example less than one hundred. Similarly, "large" elements are elements that cover one hundred samples or more. We describe how large elements are identified later in this chapter.

In the first, Small-Element Sampling stage, we sample each small element and defer sampling of large elements. This stage has two outputs. One output is the large elements that are not sampled. The other is the grid G and the partially populated field F. Because we focus on "small" elements and we know that each processor is working on an approximately equal sized subset of the elements, we know that no processor will see a large number of samples. This reasoning will be further formalized in subsection 4.3. Our implementation of the structure that stores F is able to scale its size based on how many samples have been encountered. Thus, this counteracts the problem where F can exceed

the size of primary memory, because we only have to store the samples we encounter and we will encounter only a small number of samples.

The second, Communication stage, re-distributes the output of the first stage to honor the second partitioning. The second partition is created dynamically at this point in the algorithm. By waiting until the first stage has completed, we can assign the pixels so that the maximum number of samples and elements remain on their processor of origin, minimizing communication. This technique of dynamically assigning pixels to processors to minimize total communication was also used in the hybrid scheme of [64], although their application was to surface rendering, not volume rendering.

The communication stage consists of "all-to-all" communications between the processors. Sample points are communicated among the processors, using the partition to determine their destination. When sending a large element, we first examine its bounding box, and determine which portions of the image space partition the bounding box overlaps. We then send this element to the set of processors that need to sample it in the next stage. At the end of this stage each processor contains all of the data (as either samples or elements) necessary to create its portion of the image with no further communication.

Our scheme does not take advantage of the optimization that allows contiguous samples to be pre-composited, allowing a few bytes to take their place. This optimization is not well suited to the sampling scheme in section 4.6. But the need for this optimization is greatly mitigated by the high network bandwidth on modern supercomputers. Empirically, the sampling stage dominates the algorithm, so this shortcoming is not an issue.

The third, Large Element Sampling stage samples the remaining elements (all of which are "large") and adds them to the field F. The large elements are sampled selectively. Samples outside a processor's portion of the image space partition are not examined. If an element spans two portions of the partition, then the element will be sampled entirely by the corresponding two processors, but no part of it will be sampled twice.

At the end of this stage, all elements have been sampled and the field F is fully populated. This is the output of our sampling algorithm, which is now well suited for compositing. Even though F is distributed across the processors, each ray has all of its samples on the same processor. So compositing can take place with the only necessary

communication being the collection of the pixel colors to the master processor at the end of the algorithm. The entire pipeline can be seen in figure 4.1.

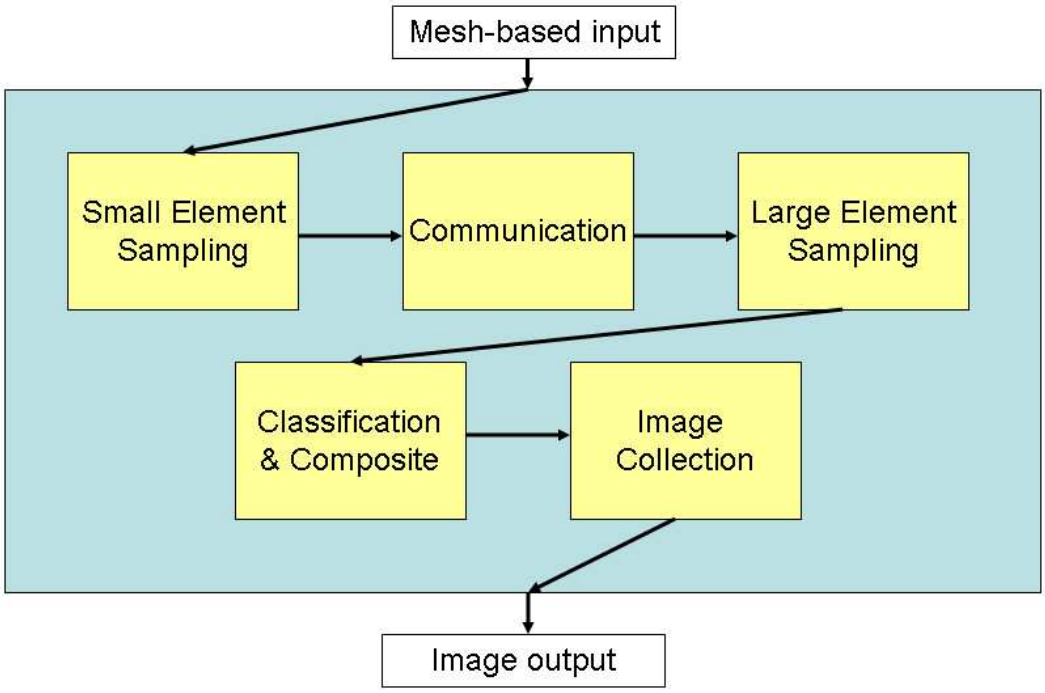

Figure 4.1: The volume rendering portion of the pipeline.

4.4.3 Load Balancing

Our goal is to maximize parallel efficiency. In this subsection, we motivate how our three-stage approach, outlined in the last subsection, helps to accomplish this goal.

Before we analyze the costs of each stage, consider the cost of doing sampling, whether it is in the first or third stage. When sampling E elements, there is an overhead with the processing of each individual element. And there is also a cost for each of the S_i sample points updated when processing element E_i. If S=$\sum S_i$, then the overhead for processing the elements is O(S + E).

How much work takes place in the first stage? Because we do not process large elements, we can easily bound this number. We do not process any elements that contain more than a constant number of samples, α. So the work in the first stage is O(S + E) = O(α*E + E) = O(E). Of course, asymptotic analysis can be misleading. In this case,

however, it gives us an actual result. There are strict limits on the amount of sampling work for each processor. In addition, by modifying α, we can control the amount of allowed discrepancy in work between the processors. Note that choices of α that are *too* small, however, cause a large number of elements to be communicated, which affects performance in the next stage.

How much work takes place in the second stage? First, consider the cost of sending a large element. When communicating a large element, the many samples it covers are *not* being communicated. If our constant, α, is large enough, then the cost to communicate large elements will always be less than the cost to communicate its samples. Following this logic, the most communication the algorithm can undergo is when communicating only samples. Although this number can be quite large, the bound on the amount of data communicated is a nice property in the context of extremely massive data sets, like the 27 billion-element simulation we discuss in the performance section.

The only work done in the third stage is the sampling of large elements. In this stage, each processor is only responsible for a limited number of samples – those that are contained within its portion of the view frustum. This bounds how much sampling work can be performed. In addition, each element in this stage is "large" and covers many samples. So the number of elements (E) will be much less than the number of samples (S): $E < S/\alpha$. So, in this stage, the amount of work per processor is also bounded: O(E+S) = O(S/α + S) = O(S).

Summarizing, for all three stages, the amount of work per stage is bounded. In addition, most processors approach these bounds, leading to good parallel efficiency. Of course, degenerate cases exist where some processors will be at their theoretical bounds while other processors spin idly. In practice, however, the amount of work per processor is relatively even.

4.5 3D Rasterization

The 3D Rasterization algorithm described in this section is an example of a sampling algorithm from section 4.3. It is similar in spirit to the 2D Rasterization algorithms

employed by graphics hardware. The key difference, of course, is that the rasterization occurs in three dimensions (over a volume), rather than two (over an image). This technique is closely related to the method described in [86].

The rasterization algorithm simply operates on each element, one at a time, updating F as it goes. For each element E, the resampler works as follows:

1. Transform E's coordinates from world space to screen space
2. Calculate E's bounding box in screen space
3. If we are deferring large elements and the bounding box of E covers too many samples, add E to the output and continue to the next element
4. Determine the set of integer depths, $\{ D_i \}$, that overlap between E and and the output grid G
5. For each D_i, slice E by D_i, resulting in polygons[2]
6. For each slice, employ a standard rasterization technique on the resulting polygons, updating the portion of the field F corresponding to D_i.

For clarity, consider the following example for a single element:

The first step is to transform the element to screen space (defined over Width, Height, and Depth axes). Assume the result is a tetrahedron, T, with points (8.5, 6.5, 4.2), (12.5, 6.5, 4.2), (10.5, 11.1, 4.2) and (10.5, 7.5, 6.8). The second step would be to calculate its bounding box: [8.5-12.5, 6.5-11.1, 4.2-6.8]. In the third step, we determine the bounding box that the element can contain no more than 40 ($4 \times 5 \times 2$) samples, so this element is not "large". Then we proceed to step 4 and determine the integer depths that overlap between T and G: 5 and 6 (see figure 4.2).

We would then slice T by D=5 (see figure 4.3). This slice results in a triangle. We then employ a standard rasterization technique to the triangle. We would start by rasterizing along Heights 7, 8, and 9. For Height=7, we find samples at Width=10 and Width=11. Height=8 also yields samples at 10 and 11. Height=9 yields no samples. The

[2]Curvilinear meshes can technically give curved polygons. We approximate these with linear polygons, similar to techniques used for isocontouring.

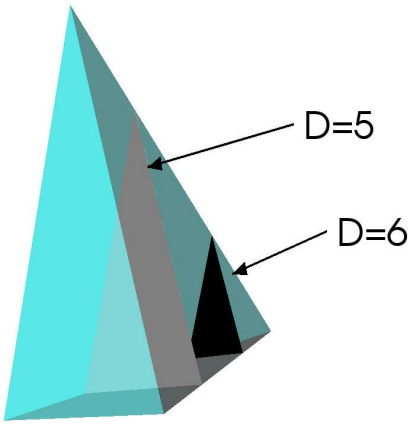

Figure 4.2: Tetrahedron T, rendered transparently in cyan. Slice D=5 is gray, D=6 is black.

entire slice at D=6 also yields no samples. In all, tetrahedron T yields 4 samples. These samples are then placed into F.

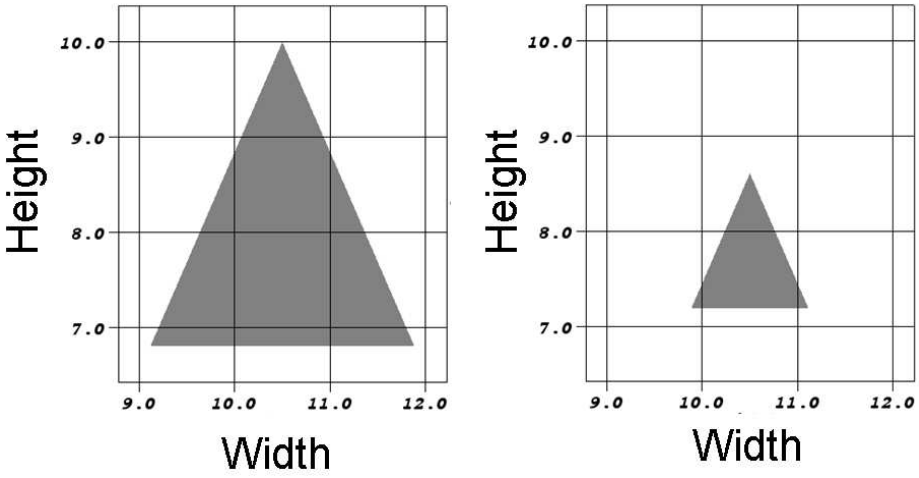

Figure 4.3: Slice at D=5 (left) and D=6 (right)

The 3D rasterization algorithm has many positive traits. First, for each element, it is possible to immediately determine the bounding box of the samples which overlap it. It is not necessary to trace rays to discover the list of cells along that ray. Second, unlike a standard projection scheme, the ordering of the resulting samples is preserved by the field F and the grid G. Third, this scheme is fairly general. It can accommodate structured grids, and curvilinear and unstructured meshes (but not point clouds). For unstructured meshes, it can accommodate the complete finite element zoo and virtually any additional element type. It can operate on multiple fields, allowing for multi-variate volume renderings. Fields

can be either element-centered (i.e. piecewise constant) or node-centered. All of these options are incorporated into our implementation.

There are also negative aspects to the rasterization scheme. First, the technique for sampling is prone to missing data. If an element does not overlap with a sample point then that element is not reflected in the resampling. This is a common case for extremely small elements and more discussion can be found in [46]. We overcome this problem by having small elements locate the nearest sample point and attempt to affect its value. This can lead to two or more elements trying to affect the same sample. In this case, we use an arbitrator that chooses the "best" sample, where "best" is chosen to be the one with the highest opacity. The second problem with this scheme is less serious. The method of examining an element's bounding box to estimate the number of samples it contains can lead to overestimates. We do not attempt to correct this problem, because it does not lead to appreciable performance degradation.

4.6 Kernel-Based Sampling

Our Kernel-Based Sampling algorithm is designed to have each data point influence a region around it. This is a natural operation with element-centered variables. The data point is placed at the middle of an element and its region is guided by its bounding box. For nodal variables, each node's neighboring elements must be examined to determine the correct size for its sampling region. In our current implementation, we operate only on element-centered variables, and we average nodal variables to element centers.

The procedure to process an element E with variable value V is:

1. Transform E's coordinates from world space to screen space
2. Calculate E's bounding box in screen space
3. If we are deferring large elements and the bounding box of E covers too many samples, add E to the output and continue to the next element
4. Calculate a maximum radius of influence for E, $R_{max}(E)$

5. For every sample within $R_{max}(E)$ of E's center, update the field F with V and a weight ω

The size of the kernel, $R_{max}(E)$, is based on the size of the element, allowing larger elements to have greater impact than smaller elements. $R_{max}(E)$ is assigned to be fifty percent bigger than the distance from the center of the element's bounding box to one corner of the bounding box. However, for small elements, $R_{max}(E)$ can be so small that it will not affect any samples. To counteract this, we never let $R_{max}(e)$ drop below the global constant, R_{min}. R_{min} is chosen so that even the smallest element will affect at least one sample.

The weights, ω, vary based on the distance to the element center. Samples closest to the element center should strongly reflect the element's value. Moving away from the element center should allow for more blending with the neighboring element values. So we set the weight to be inversely proportional to the distance to the center of the element.

The formula for weight, ω, is:

$\omega = \frac{1}{D_p+\epsilon}$ - ω_0, where

D_p = proportional distance to center of element

and ϵ and ω_0 are shape factors

For each sample, $D_p = \frac{D_s}{R_{max}}$, where D_s is the distance from the sample to the center of the element. The ϵ term eliminates division by zero, and determines the peak contribution an element can make, which occurs when the element center and a sample are coincident. When the distance from a sample to the element center is R_{max}, D_p is one. Since we would like the weight to be zero at the maximum radius, we introduce our second shape factor, ω_0, and set it to $\frac{1}{(1+\epsilon)}$.

Once each element has updated the samples, we assign the final sample value to be the weighted average. Formally, if elements E_i update a specific sample with their values V_i and weights W_i, then the final value of the sample will be $\frac{\sum W_i * V_i}{\sum W_i}$. Also, note that this calculation does not require storing all encountered samples. Instead, running totals can be kept for $\sum W_i * V_i$ and $\sum W_i$, minimizing storage overhead.

Our $\sum W_i$ term also plays an important role in establishing the boundary of the

data set. Each data point is sampled onto the region around it, regardless of the samples actual membership in the data set. Samples outside the boundary will have low $\sum W_i$ values. Reducing the opacity of these samples effectively creates the boundary in an anti-aliased way. Note that this is similar to the strategy applied in [61], although we do this correctly in the post-transformed space, where [61] did this in the pre-transformed space.

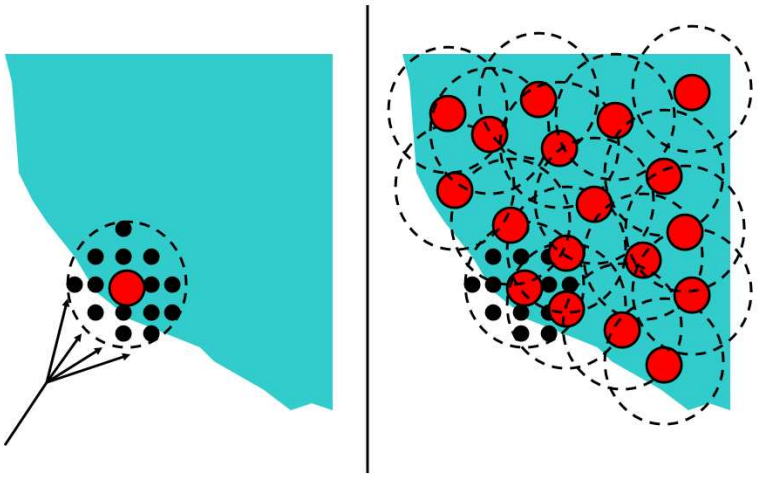

Figure 4.4: On the left, an element's (red circle) sampling region (dotted circle) includes samples outside the data set boundary (marked with the four-way arrow). These samples will be updated with the element's value, but they will not affect the final picture because their $\sum W_i$ will be low. On the right, we see how each of the samples inside the data set are affected by many elements, giving higher $\sum W_i$, while those outside the data set are affected by fewer elements.

The Kernel-Based Sampling algorithm also has many positive traits. First, like the 3D Rasterization algorithm, elements can be sampled independently and efficiently. The ordering of the resulting samples are preserved by using the field F and the grid G. Second, this scheme is also very general. In addition to handling all mesh-based data sets, this scheme is ideal for rendering point clouds. Third, unlike the 3D Rasterization algorithm, this technique consistently produces high-quality pictures and more accurately represents the entire input, including small elements.

There are also negative aspects to this scheme. First, element shapes are completely disregarded in the current implementation. Samples are updated based entirely on the element's R_{max}. In the common case for large scale data, however, an element is projected to only a few samples and this is inconsequential. Second, this algorithm processes

the same sample multiple times, one for each of the nearby elements. This contrasts with the 3D rasterization algorithm, which only operates on a given sample one time. This results, of course, in higher running time.

4.7 Scalability

We now present the scalability of the algorithm. We performed the study on a 100 million element unstructured mesh and a 1024x1024 image with 1000 samples in depth. For each processor count, we measured the elapsed rendering time with multiple options. One option was the sampling algorithm to use: 3D rasterization or Kernel-based. The other option was whether the camera was located inside or outside the data set. Tables 4.1 and 4.2 show that the algorithm was strongly scalable in all configurations that we tested.

Although we feel the algorithm has excellent scalability in this regime of processor count, we foresee some limitations as processor counts get larger. The time to create the final output image, including collecting portions from other processors and transferring the image from the server to the client for display, takes one tenth of a second. When more processors are used, this constant will begin to affect scalability. We do believe, however, that this algorithm will still be effective with larger processor counts, but as a weakly scalable algorithm. We believe the frame rate can never fall below some constant (minimally 0.1 seconds), but, given more and more processors, we feel that proportionally more data can be rendered in the same amount of time.

The timings were run on gauss, a 512-processor cluster of 2.4GHz Opterons connected by an InfiniBand network. Gauss is currently #409 on the Top500 listing of Super-Computers. For the timings, the algorithm was embedded in VisIt, rather than run as a stand-alone application. We disabled the mode that incorporates geometric primitives (for bounding boxes and other annotations), however, because delays that mode introduces are unrelated to our algorithm.

Procs	3D Rasterization/ outside	3D Rasterization/ inside	Kernel-Based/ outside	Kernel-Based/ inside
25	12.0s	21.9s	12.1s	63.7s
50	5.8s	12.1s	5.8s	30.5s
100	3.0s	7.0s	3.1s	15.3s
200	1.6s	3.6s	1.3s	7.6s
400	0.9s	2.1s	0.7s	4.1s

Table 4.1: Measuring absolute running time for different configurations and different processor counts.

Procs	3D Rasterization/ outside	3D Rasterization/ inside	Kernel-Based/ outside	Kernel-Based/ inside
25	300s	548s	303s	1593s
50	290s	605s	290s	1525s
100	300s	700s	310s	1530s
200	320s	720s	260s	1520s
400	360s	840s	280s	1640s

Table 4.2: Measuring the total running time on all processors for different configurations and different processor counts. (Total running time is calculated as the running time to produce the image multiplied by number of processors.) Our algorithm produces approximately the same number for each column, which is the trait of a strongly scalable algorithm.

4.8 Results

4.8.1 Shock Propagation in Nano-Porous Metal

Classical Molecular Dynamics (MD) simulations are a common means to study material properties at the fundamental level of individual atoms, including such phenomena as plasticity, ductile failure, brittle failure, material response to ion or micro-meteorite impacts, laser ablation and more. The simulation we use for test purposes involved 1,013,455,626 (over a billion) atoms resulting from the study of shock wave propagation through a metallic foam. In this case, the shock was introduced in a high density region to the left of figure 4.5, and has traveled partway through the porous solid, which was set up to gradually decrease in density as the shock travels to the right of the material sample. A typical variable of interest is the local energy potential of each atom, which can reveal dislocation and slip-plane

structures that arise in response to the shock passage. The transfer function was chosen to reveal these structures (seen toward the left of the image as cross-hatch like patterns), as well as the unshocked, lower-density filament structure to the right (with the red outlines).

Given there is no mesh structure for MD simulations, the kernel-based resampling is the only choice for applying volume rendering. For solid mechanics problems like this, there is a fairly tight distribution of inter-atomic spacings (distance to nearest neighbor). Typically, it is appropriate to set the resample kernel radius to be a factor of 1.5-3 times the average inter-atomic spacing, depending on whether views of fine void structures or smoother averaging is desired for the application analysis.

We visualized this data set using 256 processors. For smaller images (400x400), each rendering takes about 8 seconds. For large images (1024x1024), it takes 30 seconds.

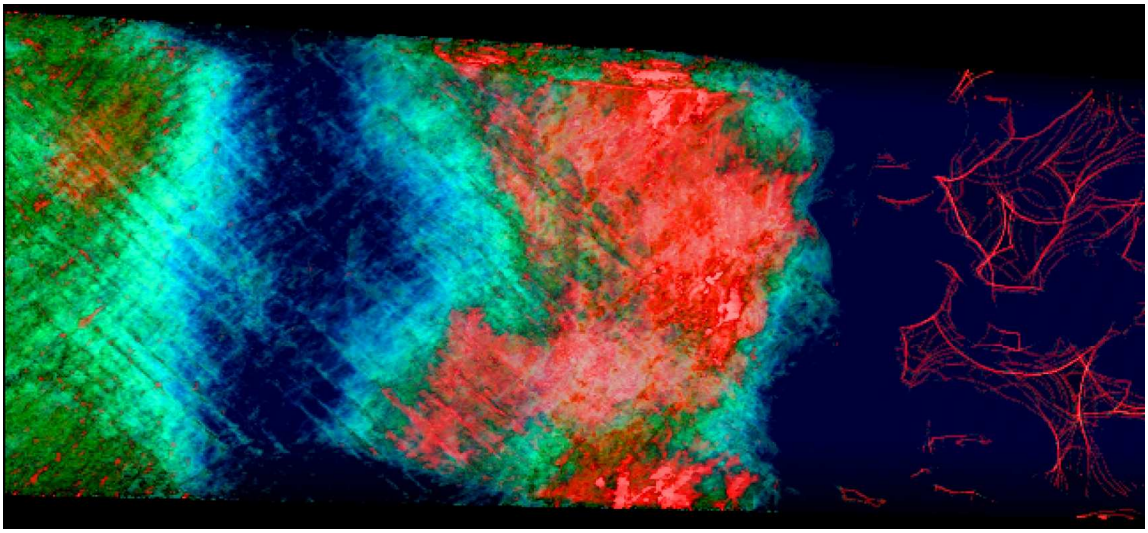

Figure 4.5: A volume rendering of over a billion atoms, rendered using the kernel-based resampling method. The simulation was produced by Farid Abraham of LLNL on 8000 processors of the ASC Purple supercomputer, in collaboration with Mark Duchaineau. A shock is traveling from left to right through the metallic foam, which was designed to gradually change from high to low density during the shock front propagation. The cross-hatch patterns on the left show the emergence of plasticity due to dislocations, while the undisturbed filament structure is highlighted on the right.

4.8.2 Rayleigh-Taylor Instability

We next applied our algorithm to a 27 billion element rectilinear grid simulation of a Rayleigh-Taylor Instability, where heavy and light fluids mix. This calculation was done

on the BG/L Supercomputer by the MIRANDA code. We used 256 processors[3] and each 1Kx1K image took 3.8 seconds to render (see figure 4.6). This data set highlights one of the problems with our 3D rasterization scheme. When using one thousand samples per ray, the 3D rasterization scheme cannot represent all of the data, since there are so many more elements than sample points.

For rectilinear grids, our 3D rasterization scheme has been optimized to efficiently find elements that overlap with samples. The purpose for transforming individual elements from world space to screen space is to quickly identify the samples an elements overlaps with. For rectilinear grids, this technique is not necessary, as the location of a sample point can be directly and efficiently calculated. As such, we revert to a simplified sampling scheme for rectilinear grids that does not transform individual elements to screen space. With this optimization, voxels that do not overlap with samples are ignored. In effect, this makes it a scheme that is indifferent to data set size, provided the data set can fit in primary memory. For any size rectilinear grid, the exact same amount of work is performed. When rendering a 1 billion element rectilinear grid, with the same number of processors, the rendering rate improved by a factor of 2.5. We attribute this unexpected speedup to increased L1- and L2-cache hits.

When we switched to the kernel-based sampling scheme, performance dropped considerably because every element had to be traversed. In this mode, it took 45 seconds to generate a picture. Because of the nature of the data set, (smooth features that span many elements), differences between the two pictures were not significant.

4.8.3 Blast Calculation

Our next calculation simulated an explosive blast wave as it passed over a section of a wall consisting of reinforced concrete. It was calculated using the ALE3D simulation code with Arbitrary Lagrange-Eulerian (ALE) methods on a 3D unstructured hexahedral mesh. For this study, we subdivided each hexahedron into five tetrahedrons, for a total of 138 million, to produce an unstructured mesh with an element count that would stress our algorithm. Because some surfaces were so thin, even the kernel-based sampler had

[3]Machine unavailability forced us to use this smaller number of processors.

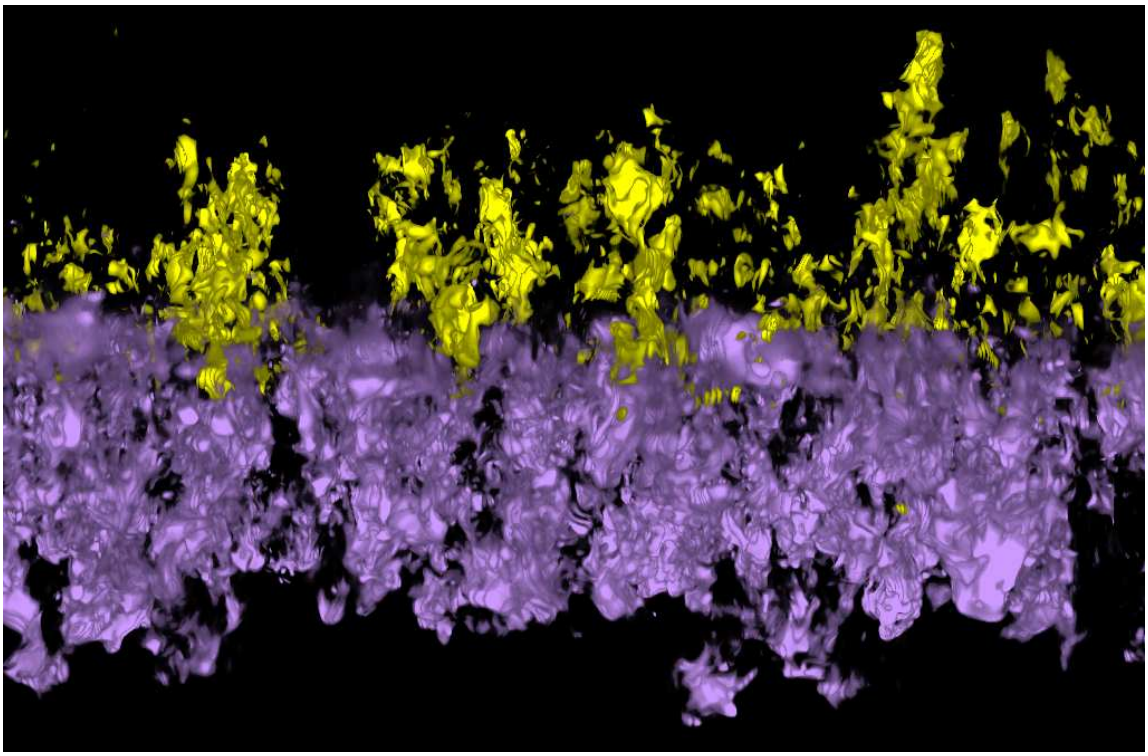

Figure 4.6: The mixing layer between heavy and light fluids. We are rendering based on vertical velocity. Yellow volumes are moving up, purple volumes are moving down.

problems properly reflecting the data. The averaging of values within a sample's region led to smearing. We increased the samples per ray to 2000, which obviously led to a degradation in performance. We used only 128 processors and were able to render a 1Kx1K frame every 35 seconds (see figure 4.7). Considering we were operating on a larger data set and had twice as many samples, this is consistent with the scaling study we performed earlier.

4.9 Future Work

There are many performance improvements that can be made to our algorithm. We did not partially composite samples before sending them, because a sample's value with the kernel-based technique can not be determined until all of the surrounding elements have made their contribution. We believe that, despite this issue, we can proceed with this optimization on a restricted basis by determining situations where samples have received their full contribution and allowing those samples to be partially composited. Another option would be to apply compression algorithms on the samples themselves before communication.

In addition, we could further reduce communication by changing the way we partition the image volume. We currently partition over the pixels, which leads to long shafts in depth. An alternative would be to create cubes by dividing the volume in depth as well. This would increase the volume to surface area ratio, minimizing the number of times a large element is mapped to multiple processors. We have experimented with implementations like this, but decided not to present this scheme, because the extra steps to composite the samples would further complicate the overall algorithm. Finally, we have considered accelerating the sampling phases by using 3D graphics hardware. Of course, this approach would only be viable on clusters where graphics hardware is available.

There are also opportunities for improvements in picture quality. We currently have no lighting model, and we would benefit from incorporating pre-integration techniques.

4.10 Summary

We have presented an algorithm that allows massive data sets to be volume rendered at interactive speeds, given adequate computing resources. The algorithm is sample-based, and the overhead for processing the samples is high. As a result, this algorithm would be a poor choice for volume rendering small data sets with low compute power, because a disproportionate amount of time is spent calculating the values of the samples. But, again, the algorithm is ideal for massive data sets and we have demonstrated good performance on some of the largest data sets ever simulated.

We caution the reader against characterizing this algorithm as a brute-force algorithm applied on a large machine to achieve a result. When scaling up to large numbers of processors, the difficulty comes in maintaining good parallel efficiency. We are able to do this elegantly with our hybrid approach. By splitting processing into small and large element sampling phases, the work performed in each phase cannot exceed known bounds. These bounds guarantee that no processor is tasked with a disproportionate amount of work to perform while other processors spin idly. This hybrid algorithm is the principal contribution of this chapter. In addition, we have introduced some sampling schemes that are well suited for this general approach and allow for either quick or accurate sampling.

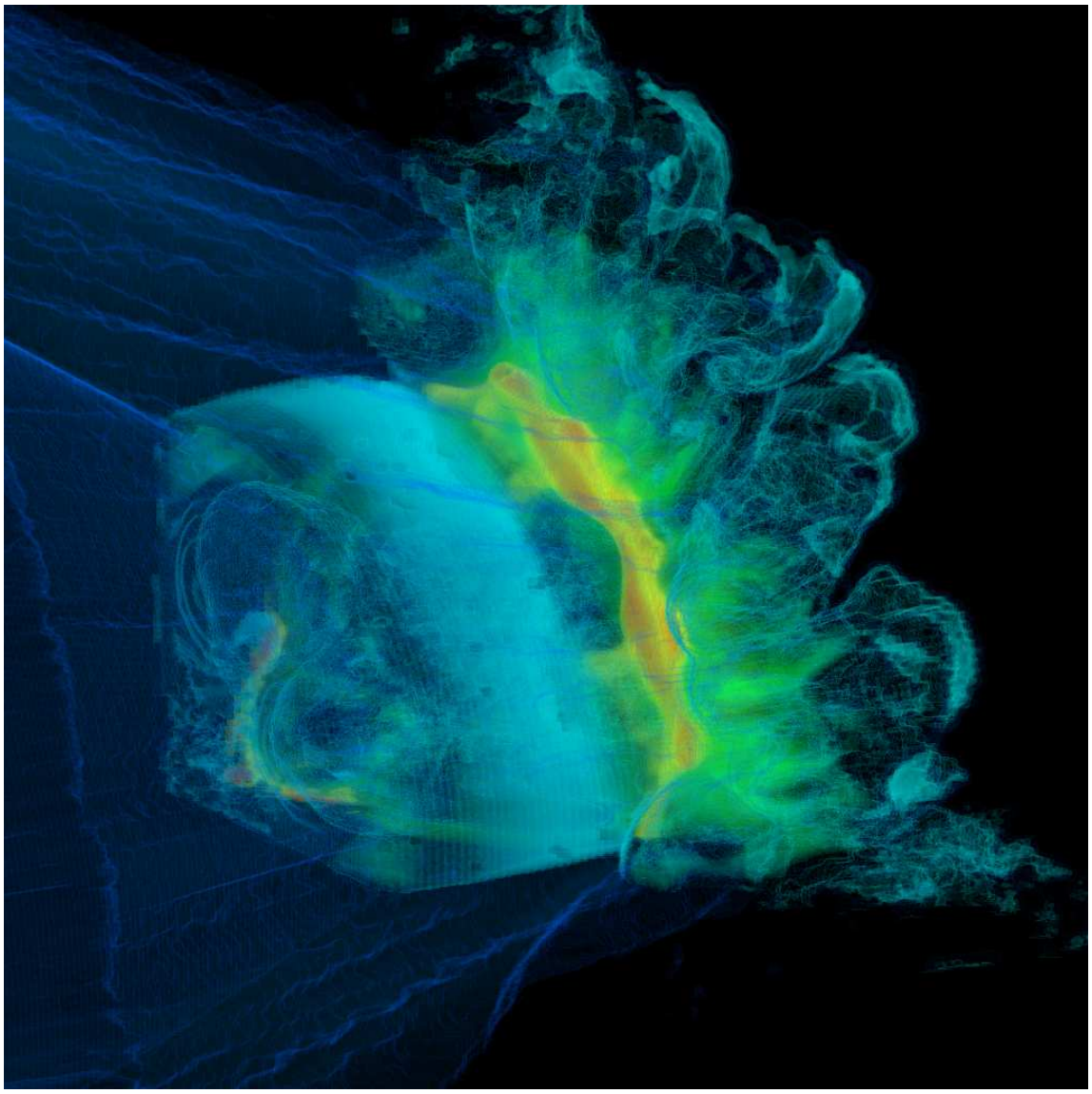

Figure 4.7: An explosive driven blast wave passing over a round section of reinforced concrete (1/4 symmetry)

Chapter 5

Comparing Data Sets

In this chapter[1], we present a data-level comparative visualization system that is very general in nature. The types of possible comparisons include comparisons in physical space, comparisons in logical space, comparisons across symmetry conditions, and comparisons to analytical functions.

Our system utilizes two key pieces of technology: (1) cross-mesh field evaluation – algorithms to evaluate a field from one mesh onto another – and (2) a highly flexible system for creating new derived quantities. Further, these capabilities are deployed in a richly featured, data flow network-based application, enabling diverse visualizations and analyses. The techniques have been fully parallelized and our results are applicable to tera-scale and even peta-scale data sets.

Where many previous comparative visualization efforts have focused on "$A - B$" comparisons, our system is able to compare many related simulations in a single analysis. In this chapter, we describe new applications of comparative techniques that allow for novel visualizations of ensembles of simulations or time-varying data.

[1]Much of the text from this chapter comes from a pending submission to the journal IEEE Transactions on Visualization and Computer Graphics. The submission was a collaboration between Hank Childs, Sean Ahern, Jeremy Meredith, Mark Miller, and Ken Joy

5.1 Introduction

One of the most common activities of an analyst is to perform comparisons. These comparisons take different forms:

- comparing a simulation to experimental data,
- comparing a simulation to a legacy simulation result,
- comparing the results of a simulation before and after key algorithmic changes,
- comparing the results of simulations with different initial geometries,
- comparing the results of a simulation as it evolves in time.

These are mostly "A-B" type comparisons, where two results are compared. Sometimes, however, the comparisons take place between a series of data sets, for example when looking at an ensemble of calculations (such as a parameter study) or looking at time-varying data.

In this chapter, we present a system for facilitating data-level comparisons. Our system incorporates multiple inputs and creates an output data set that reflects a comparison. The system combines two key technologies:

1. *cross-mesh field evaluation*, where a field from one mesh is evaluated onto another mesh, and
2. *derived quantity generation*, where new fields are created from existing ones

These two capabilities combine to form a powerful and flexible system where end users are given great latitude to create data sets tailored to application-specific comparative metrics. We have integrated this system with a richly featured, turnkey application for visualization and analysis. This allows the new, composite data set to be visualized with well-known techniques: slicing, iso-surfaces, volume renderings, etc. Our methods are capable of supporting large-scale data and we demonstrate examples where processing is done in parallel on meshes with hundreds of millions of elements.

The best comparison in design to our system is to that of Shen et al.[69, 70]. For both systems, the data sets to be compared are placed on a common mesh and derived

quantities are generated. Our system, however, contains the following extensions to this idea, all of which we believe to be new and significant:

- Comparison of more than two input data sets, which enables powerful applications for time-varying data and parameter studies[2];
- Comparison across symmetry conditions as well as to analytic functions;
- The flexibility of applying either position-based or connectivity-based comparisons;
- A richer language for defining derived quantities as well as a richer set of pre-defined operations, which in turn enables more kinds of comparisons;
- Integration with a richly featured visualization and analysis tool, allowing greater capabilities regarding how to analyze the composite data set; and
- A fully parallelized solution.

In section 5.3, we discuss the types of comparisons we have addressed. Section 5.4 presents our methods for derived quantity generation and cross-mesh field evaluations within a distributed-memory data-flow framework. Section 5.5 describes the use of comparative methods for assessing defects in as-built models, for evaluating time-varying data, and for outcome analysis with parameter studies.

5.2 Related Work

Shen et al.[69] define three basic areas of comparative visualization: *image-based*, *data-level*, and *topological* comparisons. Image-based methods compare results in image space; data-level methods compare results in object space; topological-based methods compare results in feature space.

Most image-based comparison systems [24, 85, 88] perform image differencing algorithms on images from multiple inputs. These systems are limited to comparing visualizations where the entire data set can be represented by a single image. These techniques

[2]In [70], this notion was suggested, but not discussed.

are extremely important in the context of comparison to experimental results, such as is the case with [24] and [88].

The VisTrails system of Bavoil et al.[7] coined the phrase *multiple-view comparative system* to describe image-based systems where plots are placed side-by-side or potentially overlaid. In multiple-view comparative systems, the burden is placed on the human viewer to visually correlate features and detect differences. Although the human visual system is often a great feature detector, too often the viewer can infer only gross qualitative differences in the data sets. Important differences can be too easily obscured and subtle differences are frequently totally lost.

Topological methods [5, 26, 79] compare features of data sets. Typically, they survey the data set, create summaries of the data, and develop methods to visualize these summaries.

The ALICE Differencing Engine of Freitag and Urness [27] is a data-level comparison tool. It is limited to comparing data sets that have identical underlying meshes and only allows data differencing comparisons. Shen et al.[69, 70] place data sets on an intermediate mesh and utilize derived quantities to form comparisons. As discussed in the introduction, this is the approach most similar to our own. Sommer and Ertl [72] also base their comparisons on data-level methods. Their system employs only connectivity-based differencing, although they also consider the problem of comparisons across parameter studies.

A separate area of related work is that of derived quantity generation. Many visualization systems [1, 21, 35] provide subsystems for generating derived quantities. Moran and Henze give an excellent overview in [54] of their DDV system, using a demand driven calculation of derived quantities. McCormick et al. furthered this approach with Scout [50] by pushing derived quantity generation onto the GPU.

5.3 Data-Level Comparison Methods

Our process can be summarized as performing cross-mesh field evaluations and creating derived quantities. Formally, let $M_1, M_2, ..., M_k$ denote a set of input meshes, each containing a corresponding field $F_1, F_2, ..., F_k$, and let M_C denote some common mesh.

The cross-mesh evaluation step evaluates each field F_i, for $i = 1, ..., k$, onto M_C, creating resampled versions of the input fields, $F_{C,i}, i = 1, ..., k$ that represent each of the original fields on the common mesh. We then employ a function, $\Psi(F_{C,1}, F_{C,2}, ..., F_{C,k}) \rightarrow \Re^n$, that takes elements of the k fields as input and produces a new, derived quantity. The resulting field, F_Ψ, defined over M_C, is then visualized with conventional algorithms. Using this framework, we can characterize most previous systems as considering only M_1 and M_2 and visualizing $F_\Psi = F_{C,1}$ - $F_{C,2}$.

The remaining questions are: (1) What M_i's and F_i's can be used?, (2) What is M_C?, (3) How is $F_{C,i}$ generated?, and (4) What is F_Ψ?

5.3.1 What M_i's and F_i's can be used?

M_i and F_i can come from simulation or experimental observation. Rectilinear, curvilinear, unstructured, and AMR meshes are supported. The fields can be scalars, vectors, or tensors. Some common examples are: two or more related simulations at the same time slice, two or more time slices from one simulation, a single data set to be compared with itself through a symmetry condition, and a data set with an analytic function.

5.3.2 What is M_C?

Our system is general enough to use any mesh as the common mesh, M_C. It can be either a new mesh or one of the M_i. In practice, we frequently use the latter option. However, we also support generating new arbitrary rectilinear grids with a user-specified region and resolution. Further, it would be straightforward to add other methods of creating new meshes, like weaving together the input meshes or tetrahedralizing their vertices, but these techniques seem less immediately beneficial.

5.3.3 How is $F_{C,i}$ generated?

There are two choices for generating $F_{C,i}$: the evaluation can be made either in a position-based fashion or in a connectivity-based fashion.

The most common method is by position. Here, the meshes are overlaid and overlapping elements from the meshes are identified. Interpolation methods are applied

to evaluate F_i onto M_C. This technique is difficult to implement, especially in a parallel, distributed memory setting, because M_C and M_i may be partitioned over the processors differently, which requires a re-partitioning to align the data. This re-partitioning must be carefully determined to ensure that no processor exceeds primary memory. Also, it is important to perform interpolations that do not introduce artifacts, as it is unlikely that the individual elements of M_C and M_i will perfectly overlap spatially. Issues with overlaying and interpolation are discussed in section 5.4.2. Finally, note that symmetry-based evaluations are highly related to position-based evaluations, with the difference being an additional transformation that aligns data sets along the symmetry condition.

The other important evaluation technique is connectivity-based, which requires that both the input mesh, M_i, and the common mesh, M_C, are *homeomorphic*. That is, they have the same number of elements, E, and nodes, N, and the element-to-node connectivities for all elements are the same. This is often the case if M_C is selected from one of the M_i, because for comparisons across time and/or parameter studies the remaining $M_{j:j\neq i}$ typically have the same underlying (i.e. homeomorphic) mesh. In this case, the value of element E or node N in M_i is directly transferred to the corresponding element or node in M_C. Connectivity-based cross-mesh field evaluation is substantially faster; the difficult task of finding the overlap between elements and then correctly interpolating based on the overlap is eliminated entirely.

For Eulerian simulations (where node positions are constant, but materials are allowed to move through the mesh), connectivity-based evaluation yields the same results as position-based but with higher performance and is the preferred approach. For Lagrangian simulations (where materials are fixed to elements, but the nodes are allowed to move spatially), position-based evaluations are used most often, because connectivity-based evaluations generally do not allow analysts to observe differences based on spatial position. However, connectivity-based evaluation still play a role, since they allow for new types of comparisons along material boundaries, even if they are at different spatial positions. Further, connectivity-based evaluations allow for simulation code developers to pose questions such as: how much compression has an element undergone? (This is the volume of an element at the current time divided its volume at initial time.)

5.3.4 What is F_Ψ?

There are limitless forms of derived quantities, F_Ψ, that are necessary for different types of comparisons in different situations. We list a few of the most frequently used in table 5.1 as examples. These examples focus on scalar fields, so we emphasize that vectors and tensors also are supported.

	Description	Definition
1	Difference	$F_2 - F_1$
2	Relative difference	$(F_2 - F_1)/(F_2 + F_1)$
3	Maximum or minimum	$(F_2 > F_1 \ ? \ F_2 : F_1)$ $(F_2 < F_1 \ ? \ F_2 : F_1)$
4	Determine the simulation containing the maximum or minimum	$(F_2 > F_1 \ ? \ 2 : 1)$ $(F_2 < F_1 \ ? \ 2 : 1)$
5	Average	$(F_2 + F_1)/2$

Table 5.1: Common derived quantities for comparisons. For simplicity, we assume only two meshes, M_1 and M_2. The common mesh (M_C) is M_2, and the fields from the evaluation phase are F_1 and F_2. #1 allows users to explore the differences between the two data sets. #2 is used in a similar fashion, but amplifies small changes on relatively small quantities. #'s 3-5 (minima, maxima, and averages) are useful when going beyond simple "$A - B$" comparisons. When analyzing time series or parameter studies with many related inputs, these quantities allow for visualization of all data sets with a single, composite data set.

5.4 System Description

Three critical pieces make up the comparative system: the derived quantity system, the cross-mesh field evaluation methods, and the greater system that allows a distributed, parallelized implementation. The greater system has already been described in chapters 2 and 3 so its description will be omitted here.

There are other components of the system that have lesser importance. We enumerate them here only for completeness and do not discuss them further: We have provided an interface that allows the user to manage the entire comparison process, including what data sets are compared, how, and onto what mesh. We have provided symmetry operators that reflect a data set across a plane, a line, or a point, as well as arbitrary affine transformations (specified by a 4x4 matrix). We have the ability to evaluate an analytic function

on any mesh. Finally, we have routines to construct high-resolution rectilinear meshes, to serve as the common mesh, M_C.

5.4.1 Derived Quantities

There are two key areas to the derived quantity system. One is the expression language that allows end users to create new, arbitrary derived quantities. The other is the mechanism that transforms an instance of an expression into a form suitable for data flow networks.

Expression Language

We have developed a functional, string-based system to allow users to create new derived quantities. This results in an expression language syntax that enables users to compose derived quantities in arbitrary ways. A major goal of the design of our expression language was to provide an intuitive interface where creation of new derived quantities required little to no learning curve for common operations. For example, the average of two fields, A and B, is as simple as "$(A+B)/2$".

Of course, users will want to create derived quantities that are more than simple mathematical constructs. Support exists in the language for composing scalar quantities into vectors or tensors, and for extracting scalar components back out. Notation for strings, lists, ranges, and strides allows selection of materials, parts, and other subsets of elements. For accessing other files, either within the same sequence or in different sequences, the language supports references by cycle, absolute and relative time index, and filename. Other named operations are referenced as functions, and a small selection of the over one hundred available are listed in table 5.2.

The strength of our expression language lies in the richness of functionality we have provided and the interoperability between these expressions. Consider, for example, computing divergence ($\bigtriangledown$). If a two-dimensional vector F is defined as $P\hat{x}+Q\hat{y}$, then $\bigtriangledown F = \frac{\partial P}{\partial x} + \frac{\partial Q}{\partial y}$. A user can calculate divergence directly using the built-in function, divergence(). But, for illustrative purposes, it is also straightforward to calculate divergence using other functions as building blocks: "divF = gradient(P)[0] + gradient(Q)[1]". We created a

Math	+, -, *, /, power, log, log10, absval, ...
Vector	cross, dot, magnitude
Tensor	determinant, eigenvector, effective (e.g. strain), ...
Mesh	coordinates, polar, volume, area, element_id, ...
Field Operator	gradient, divergence, curl, Laplacian
Relational	if-then-else, and, or, not, $<, \leq, >, \geq, =, \neq$
Mesh Quality	shear, skew, jacobian, oddy, largest angle, ...
Trigonometric	sine, cosine,..., arctangent, degree2radian, ...
Image Filters	mean, median, conservative smoothing
Miscellaneous	recenter, surf. normal, material vol. fraction, ...

Table 5.2: Our expression language allows for all of these functions to be combined in arbitrary ways.

custom scanner and parser that constructs a parse tree based on expressions like this one. Figure 5.1 contains the parse tree for this divergence expression.

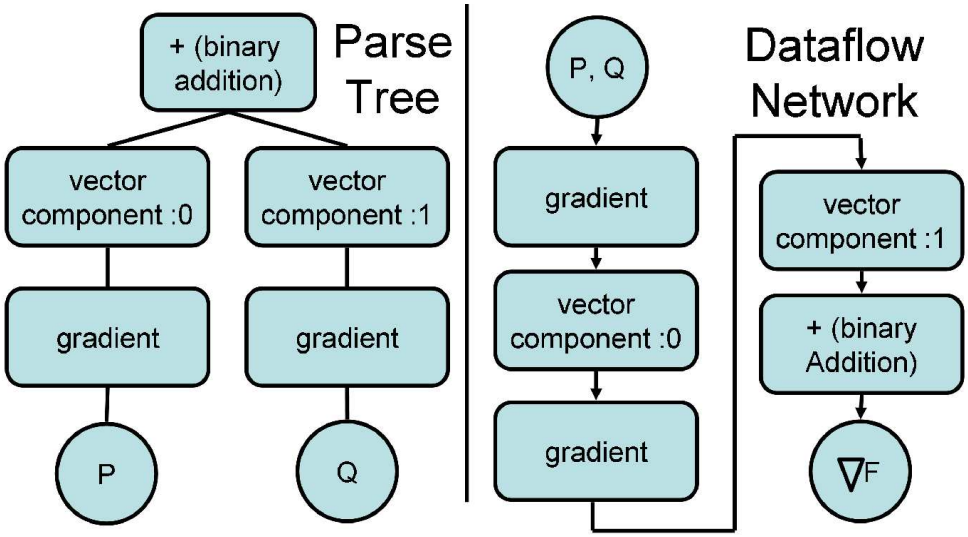

Figure 5.1: Creation of a derived quantity. On the left, we see an expression's parse tree. On the right, we see the linearized data flow network for that parse tree.

Data Flow Networks

When assembling a data flow network, the Expression Evaluator Filter (EEF) is used to construct derived quantities. The EEF is typically inserted as the first filter in the pipeline, immediately following the file reader module. In a preparatory phase, the other filters place the names of their required variables into a list. When executing, the EEF

cross-references this list with known expression names to determine what derived quantities need to be calculated.

The EEF dynamically creates a sub-network to create needed derived quantities. To do this, it first consults the parse trees of the expressions. For each node in each parse tree, a filter that can perform the corresponding operation is placed into the sub-network. Ultimately, this sub-network reflects a linearized form of the parse trees. The linearization process requires the EEF to do dependency checking between all of the parse trees for all of the expressions involved to ensure that every filter has the inputs it needs. Our implementation supports the linearization of *any* parse tree, including this dependency checking. An example of the pipeline from the divergence expression from the previous subsection is also in figure 5.1.

Our systems support the accumulation of partial results onto the common mesh M_C so that individual $F_{C,i}$'s can be quickly discarded. While the sub-network is executing, the EEF is able to determine when intermediate variables are no longer needed and remove them. Through this mechanism, the EEF is able to successfully handle many related data sets that would otherwise exceed the available memory.

Cross-mesh field evaluation is related to derived quantity generation because, from the perspective of the common mesh, it results in the creation of a new field. So it is natural to expose cross-mesh field evaluation capabilities as expressions. To perform a cross-mesh field evaluation, the user simply defines an expression involving built-in functions for the evaluation algorithms – position-based, connectivity-based, and symmetry-based. (Comparisons to analytic functions do not merit a filter in this context because there are no "cross-mesh" comparisons.) Like all other expressions, the comparison expressions have corresponding filters that can be placed in the sub-network to perform the cross-mesh evaluation. By combining these algorithms with expressions, users can direct the creation of new, derived quantities from a multitude of sources on the same common mesh. Furthermore, they can manipulate these quantities using all of the previously mentioned expressions to create interesting comparisons.

Finally, although derived quantity generation typically takes place immediately after reading the data, it is also possible to defer their evaluation until later in the pipeline.

This ability allows for the common mesh to be transformed before the cross-mesh field evaluation takes place. This is important when registration is needed, for example for comparison with experimental data.

5.4.2 Cross-Mesh Field Evaluation

The implementations of the various filters for cross-mesh field evaluation (CMFE) are similar. They all have one input for the common mesh and they all are capable of dynamically instantiating an additional data flow network to obtain M_i and F_i. The differentiating point between the CMFE filters is how they evaluate the fields.

The connectivity-based CMFE algorithm is the simplest. For each element or node, it places F_i onto M_C to create $F_{C,i}$. The only subtlety is guaranteeing that the partitioning of the input data (in a parallel setting) is done so that each processor operates on the same chunks of data. This is easily accomplished through the contract-based system.

The symmetry-based CMFE algorithms are highly related to the position-based CMFE algorithms. In fact, they use the same underlying evaluation algorithms. The only difference is that the symmetry CMFE filters apply a transformation to M_i before evaluation.

The position-based CMFE algorithm is complex. There are three major challenges:

1. Identifying which elements of the input mesh M_i overlap with an element in the common mesh M_C. This is called the *overlay* step.

2. Fitting an interpolant for the field on the common mesh M_C such that it matches, as closely as possible, at key points on the input mesh M_i. This is called the *interpolation* step.

3. Managing the distribution of data to maximize parallel computational efficiency in a distributed-memory environment.

The Overlay Step

We use interval trees [23] to efficiently identify elements from meshes M_i and M_C that overlaps spatially. We start by placing all of the elements from M_i into the interval

tree. Then, for each element of M_C, we use its bounding box to index the interval tree and find the list of elements from M_i with overlapping bounding boxes. We examine this list to find the elements that truly overlap (as opposed to only having overlapping bounding boxes). If M_C contains N_C elements and M_i contains N_i elements, then the time to generate the tree is $O(N_i \log(N_i))$ and the time to locate the elements of M_i that overlap with an element of M_C is $O(\log(N_i) + \alpha)$, where α is the number of elements from M_i returned by the search. This gives a total time of $O((N_C + N_i)*\log(N_i))$. Note that α is amortized out for all but degenerate mesh configurations.

Field Interpolation

The second challenge, performing credible interpolation[3], is difficult and we present only a partial solution. For each position on the common mesh M_C, we evaluate the field on the source mesh M_i at the same position. For element-centered quantities, the positions are the element centers of each element in M_C. For node-centered quantities, the positions are the nodes of M_C. We then assign the value at the position in M_i to the same position in M_C. More accurate interpolations are usually performed by tools called "linkers," which represent man-years to man-decades worth of work. Existing "linkers" have flaws: they are very slow and the interpolations they perform are typically targeted toward specific scientific domains (at the cost of others). We have designed our system in such a way that we can easily incorporate functionality from a production "linker" if one becomes available. Our algorithm was chosen because it favors performance over accuracy. We feel a good improvement to our implementation, however, would be to add the use of weighted averaging with weights based on volume overlaps.

Parallel Implementation

The final piece of the problem is to perform cross-mesh field evaluations in a parallel, distributed-memory environment. The key issue deals with spatial overlap. When a processor is evaluating a field from mesh M_i onto the common mesh M_C, it must have access

[3]Credible interpolation is a somewhat amorphous term because it varies over scientific domains. One definition may be that an interpolation of a density field be mass-preserving. Another favor may favor preserving sharp boundaries

to the portion of M_i that overlaps spatially with the portion of M_C it is operating on. There can be no expectation that the decompositions of the two meshes are spatially overlapping. Our strategy for this issue is to create a spatial partition to guide re-distribution of both of the meshes for the evaluation phase. Unfortunately, the spatial partition must be created with great care. If the partition divides space into regions that cover appreciably different numbers of elements, it will lead to load imbalances and potentially exhaust memory. Therefore, we focus on creating a *balanced spatial partitioning*, where "balanced" implies that every region contains approximately the same number of elements, E_t (see figure 5.2). The E_t elements from each region may contain different proportions of elements from M_C and M_i; in general, it is not possible to have this proportion be fixed and E_t be equal on all processors.

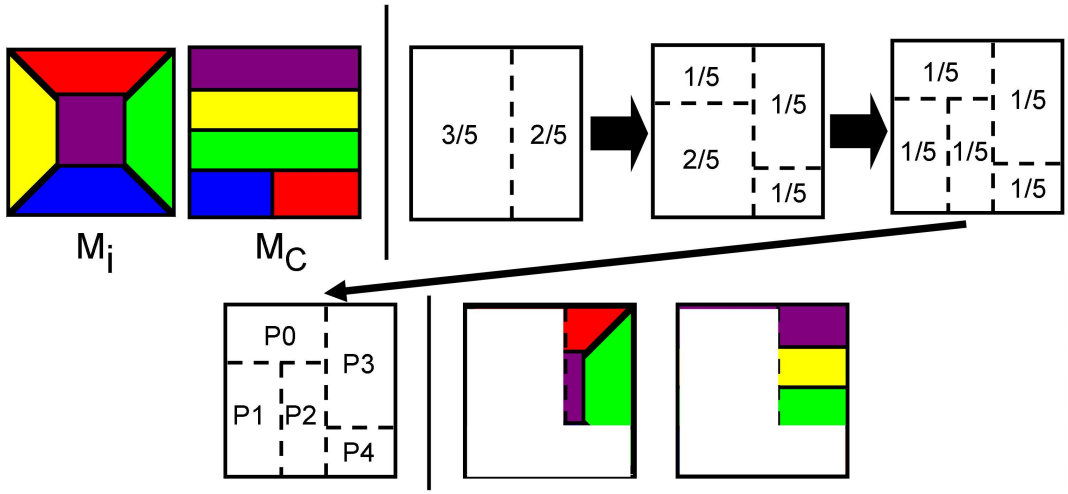

Figure 5.2: In the upper left, two meshes, M_i and M_C, are shown. Assume the red portions are on processor 1, blue on 2, and so on. We employ an iterative strategy that creates a balanced spatial partition. We start by dividing in X, then in Y, and continue until every region contains approximately $1/N^{th}$ of the data, where N is the total number of processors. Each processor is then assigned one region from the partition and we communicate the data so that every processor has all of the data for its region. The data for processor 3 is shown in the last set of figures.

The algorithm to efficiently determine a balanced spatial partitioning is recursive. We start by creating a region that spans the entire data set. On each iteration and for each region that represents more than $1/N^{th}$ of the data (measured in number of elements covered), we try to select "pivots", possible locations to split a region along a given axis. This axis changes on each iteration. Within an iteration, a small number of pivots are

chosen (typically 5) and spaced evenly among the interval of possible values. All of the elements are then traversed, and their positions with respect to the pivots are categorized. Some elements may span the pivot. In this case, the element is assigned to the same side of the pivot as its center. If a pivot exists that allows for a good split[4], then the region is split into two sub-regions and recursive processing continues. Otherwise we choose a new set of pivots, whose choice incorporates the closest matching pivots from the previous iteration as extrema. If a good pivot is not found after some number of iterations, we use the best encountered pivot and accept the potential for load imbalance.

The implementation of this algorithm is complicated by doing many parallel pivot locations at one time. The above procedure, if performed on a single region at a time, would have a running time proportional to the number of processors involved. This, obviously, is unacceptable. To overcome this, we concurrently operate on many regions at one time. When iterating over a list of elements, we avoid the poor strategy of interacting with regions that do not even contain the element. Instead, we employ a separate interval tree that stores the bounding boxes of the regions. Then, for each element, we can quickly locate exactly the regions that element spans. This variation in the algorithm gives a running time proportional to the logarithm of the number of processors, which is more palatable.

As previously mentioned, our notion of a balanced spatial partitioning only guarantees that the total number of elements from both M_C and M_i are approximately equal. Note that our interval tree-based approach would give the best results if the number of elements from M_i are balanced as well. We ignore this issue, because our larger goal is to ensure that no processor exhausts primary memory.

After the best partition is computed, we create a one-to-one correspondence between the regions of that partition and the processors. We then re-distribute M_i, F_i, and M_C with a large, parallel, all-to-all communication phase. If elements belong to multiple regions, they are sent to all corresponding processors. After the communication takes place, evaluation takes place using the interval-tree based identification method described previously. Finally, all of the evaluations are sent back to the originating processor and placed

[4] "Good" is defined as producing a desired ratio, such as one half, or, in the example in figure 5.2, two-fifths and three-fifths.

on M_C.

We used the data set from the applications section 5.5.1 for a rough illustration of performance. The evaluation is of a 1.5 million element unstructured grid onto a 1K x 1K x 676 rectilinear grid, in parallel, using eighty processors. The most expensive phase is evaluation. In this phase, each processor is doing nearly ten million lookups on its interval tree, which justifies its non-interactive running time. Table 5.3 summarizes the times spent in different phases of the algorithm. The inclusion of this information clearly does not serve as a performance study, which will be studied further in the future. However, it does inform as to the general running time for large problems.

Phase	Establishing Spatial Partition	Communi-cation of Data	Building Interval Tree	Evaluation
Time	0.7s	2.9s	5.2s	27.4s

Table 5.3: The time spent in the different phases of the parallelized cross-mesh field evaluation algorithm. The communication column represents communicating data to create the balanced spatial partitioning, and also the time to return the final evaluations.

5.5 Applications

5.5.1 As-Built Modeling

Simulations are often run on meshes that represent simplified versions of the real world objects they simulate. Details due to manufacturing processes are not often considered. As-Built Modeling [38] is an effort to incorporate these manufacturing details into the simulation process. Instead of running a simulation of an object "as designed," they run a simulation of the object "as built." Of course, this requires the creation of a mesh that includes manufacturing details. Once such data is available, analysts then wish to compare results from the "as designed" and "as built" simulations. This leads to two key questions our comparative visualization and analysis techniques help to answer. One is: how does the "as built" mesh compare to the real world object from which it is generated? The other is: how do differences in an object's "as designed" and "as built" forms lead to differences in

their respective simulations?

Assessing Errors in the As-Built Object

Here we describe how our system was applied to assess the error in how well the "as built" mesh represented the real world object. To better describe the errors with mesh creation, we now describe the underlying process of obtaining the as-built geometry. First, a part (in this case, a cylindrical phantom part shown in figure 5.3) was CT imaged. This resulted in a 676 million data point regular grid with intensity information. Next, segmentation was performed (by grouping regions with similar intensity), the boundaries of each material were extracted, and then decimated. The decimation step was necessary to reduce the number of facets to a size suitable for input to a mesh generation tool. The decimation phase reduced the number of facets from almost one million to nearly 250,000. Once decimated, the surface was imported into a mesh generation tool and an unstructured mesh of 1.5 million hexahedra was produced.

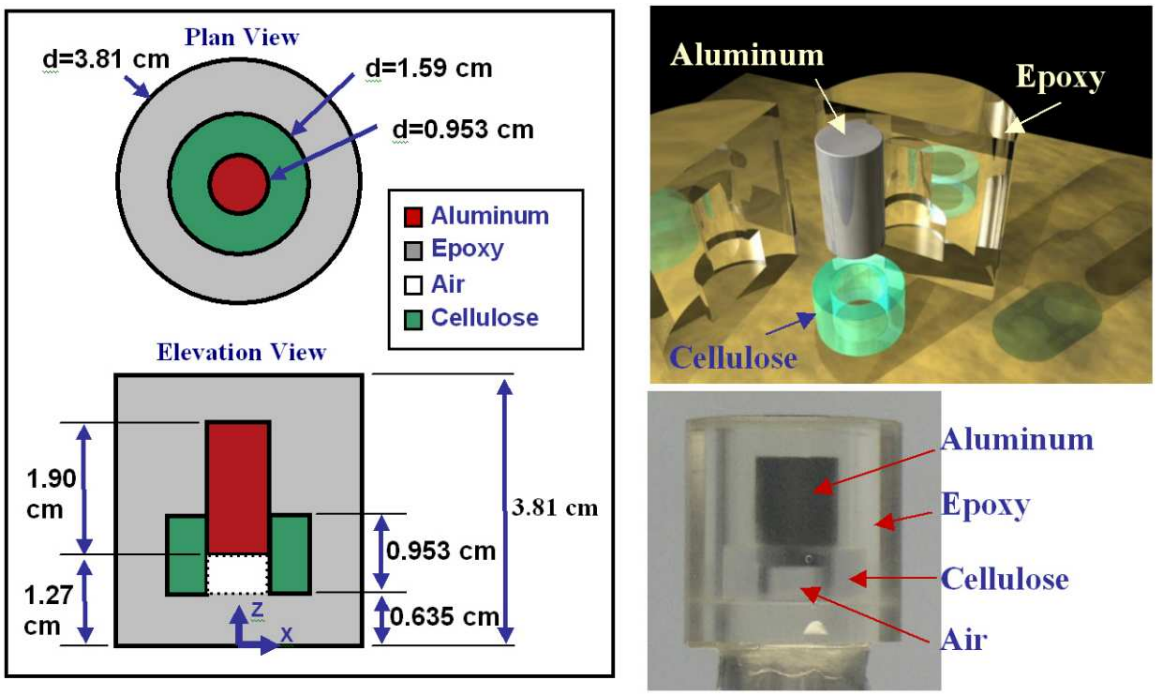

Figure 5.3: A phantom cylindrical model of four materials: aluminum, epoxy, air, and cellulose. Not represented is the glue material, which exists for the as-built part, but is not included in the as designed model. On the left is the design schematic, on the upper right is an illustrative rendering, and on the lower right is a photograph of the actual (as-built) part.

Our data-level comparative techniques cannot help quantify the errors associated with the CT or segmentation processes. But they can be used to quantify the differences between the segmented CT data set and the output of the mesh generation tool. We measured this error by measuring the volume overlap between the data (shown in figure 5.4), using a position-based CMFE. For each part, the new, derived field was a constant "one" where there was overlap and a "zero" where there was no overlap. This was accomplished through an option with the position-based CMFE to set a constant value in non-overlapping regions (in this case, zero). After calculating the volume of the segmented CT part, we removed the regions with no overlap. We then calculated the volume of the resulting data set. The resulting volume overlaps (shown in table 5.4) for aluminum was good overall, while the glue and air regions had poor overlap (due to a displacement that occurred during the meshing process).

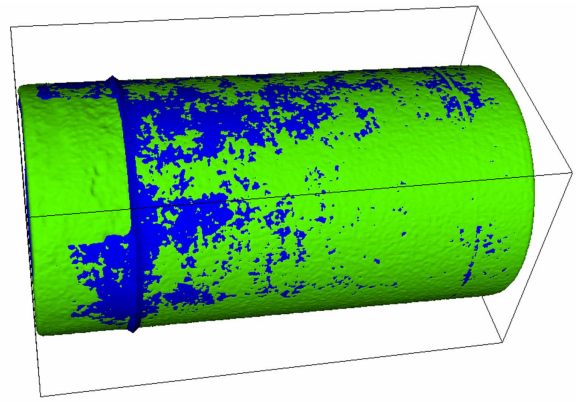

Figure 5.4: The segmented CT data (green) and the as-built geometry (blue), which were used for the volume overlap in table 5.4.

Part	Volume as segmented CT	Volume overlap	% agreement
Aluminum	1.344	1.312	97.6%
Air	0.457	0.401	87.7%
Glue	0.022	0.014	63.6%

Table 5.4: The volume overlaps of several parts. Epoxy and Cellulose are not represented, because they are not distinguishable using X-ray CT imaging.

Differences In Initial Meshes of As-Built and As-Designed Objects

The next analysis question was to determine the scope of as-built defects in the initial geometries. Using a similar technique to our CT data comparison, we calculated volume overlaps between the as-built and as-designed initial geometries. The results (shown in table 5.5) help justify the interest in as-built modeling. Overlaps were less than 95% for most materials (this is considered quite low), although some of this difference can be attributed to the as-built modeling process itself.

Part	Volume As-Built	Overlap With As-Designed	% agreement
Aluminum	1.33	1.25	94.0%
Air	0.407	0.396	97.3%
Epoxy	40.42	40.27	99.6%
Cellulose	1.18	1.08	91.5%
Glue	0.05	0	0%

Table 5.5: The volume overlaps of the initial geometries for the as-built and as-designed simulations. The epoxy and cellulose materials were differentiated through multi-modal analysis that incorporated ultrasonic scans (see [38]).

Differences In Behavior of As-Built and As-Designed Objects

The final question was to assess the differences between the as-built and as-designed models during the simulation process. For both geometries, a shock was placed at the top of the epoxy and the effects of that shock were simulated for ten microseconds. We visualized the differences in effective plastic strain, because it is a cumulative quantity, and differences between the two calculations at the final time slice are effectively integrated over all previous times. In figure 5.5, differences between the as-built and as-designed models are clearly seen at the contact points between the materials, throughout the cellulose, and inside the aluminum, especially parallel to the top of the cellulose.

5.5.2 Rayleigh Taylor Instabilities

Rayleigh-Taylor instabilities simulations model the mixing of heavy and light fluids. For this study, we looked at two types of related data sets. First, we observed a single

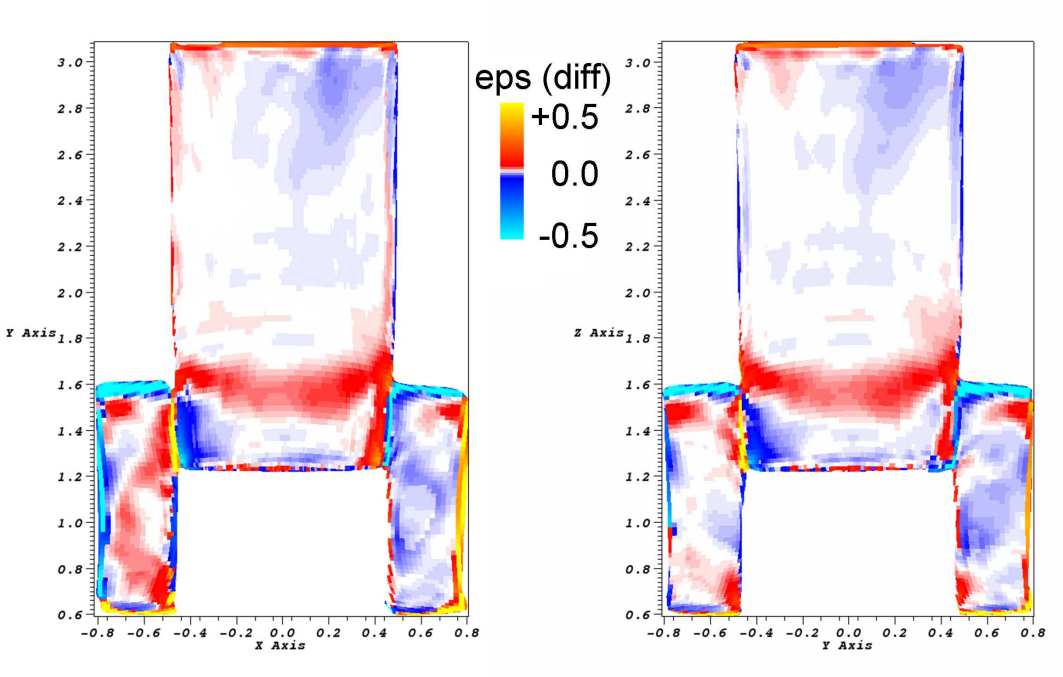

Figure 5.5: Plotting difference in effective plastic strain for the aluminum cylinder and the cellulose. The two plots correspond to two different cross-sections of the object.

simulation and its evolution in time. Then we looked at a parameter study, where turbulence parameters were varied to study how differences in these parameters affected the results. Both sets of simulations were two-dimensional calculations and performed by the Lawrence Livermore simulation code ARES.

Time-Varying Data

We started our analysis by looking at a single Rayleigh Taylor instability calculation that simulated ten microseconds, creating eighty-five time slices. Rather than focus on the differences between two time slices, we created visualizations that would summarize the whole data set. In particular, we were interested in summaries derived from a given binary condition, B_C(P, T), where B_C(P, T) is true iff condition C is true at point P and time T. For a given point P, the derived quantity was:

{ time(P): $B_C(P, time)$ AND $(\neg\exists t' : t' < time$ AND $B_C(P, t'))$ }

This derived quantity is a scalar field that, for each point P, represents the first time that $B_C(P, T)$ is true. For our study, since we were observing the mixing of two fluids,

we chose $B_C(P,T)$ to be whether or not mixing between the fluids occurs at point P at time T[5]. From figure 5.6, we can see that the mixing rate increased as the simulation went on (because there is more red than blue in the picture). We comment that the technique demonstrated here, showing a plot of the first time a binary condition is true in the context of time varying data, is very general. Further, we believe this is the first time that it has been presented in the context of creating these plots of this form (by using of data-level comparative techniques).

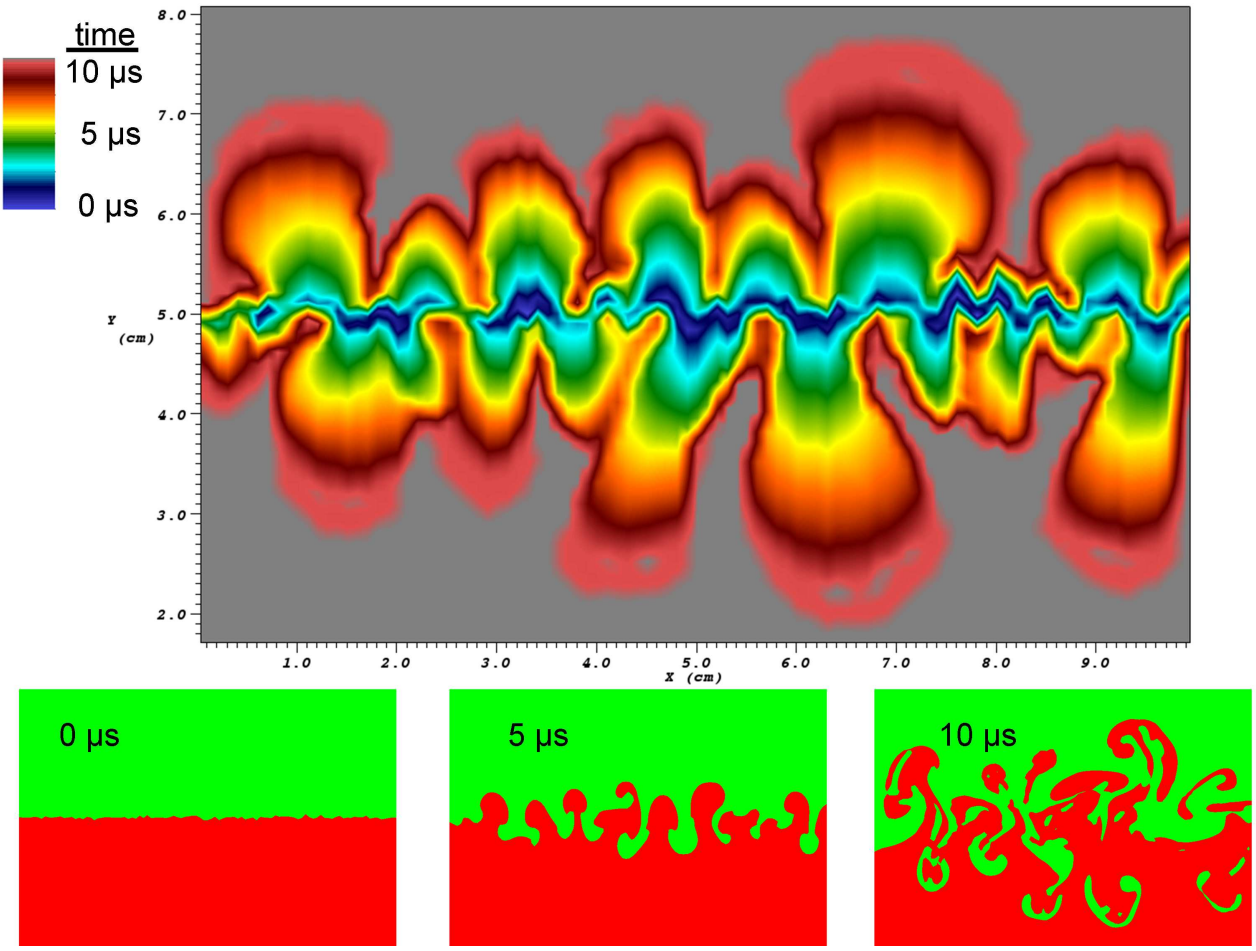

Figure 5.6: Along the top, we see a visualization comprising all time slices. Blue areas mixed early in the simulation, while red areas mixed later. Gray areas did not mix during the simulation. There is some gray inside the bulges at the bottom of the plot, because the simulation's output frequency was not high enough to measure the changes in those areas. This plot allows us to observe mixing rates as well. Along the bottom, we include three time slices for reference. Heavy fluids are colored green, light fluids are colored red.

[5]A point is defined to be mixed if the element containing that point ever contains multiple materials or if the element is clean, but the material it contains differs from the material for that point at the initial time.

Parameter Studies

A simulation of a Rayleigh-Taylor instability is dependent on certain coefficients, which are adjusted in different situations. An important question is to understand how variation in these coefficients affects the outcome of a simulation. These effects can be monitored during parameter studies, where these coefficients are varied and the results are compared. For this parameter study, two coefficients were varied independently: the coefficient for turbulent viscosity and the coefficient of buoyancy. For each coefficient, five values were chosen. Twenty-five calculations were then performed, one for each pair of coefficients.

We focused on differences in magnitude of velocity, i.e. speed. This quantity had the most variation throughout the simulations and we wanted to characterize the relation between speed and the coefficients. We examined three different derived quantities defined over the whole mesh. The first quantity was the index of the simulation that resulted in the maximum speed at the given point. The second and third, respectively, were the coefficients of turbulent viscosity and buoyancy corresponding to that simulation index.

If "sid" is a simulation identifier, $C_{tv}(sid)$ and $C_b(sid)$ are the turbulent viscosity and buoyancy coefficients for "sid", then the derived quantities, for each point P are:

1. $maxsid(P) = \underset{sid \in all\ 25}{\arg\max}\ speed_{sid}(\mathrm{P})$
2. $C_{tv}\_of\_max\_speed(P) = C_{tv}(maxsid(P))$
3. $C_b\_of\_max\_speed(P) = C_b(maxsid(P))$

The results of these derived quantities are displayed in figure 5.7. From the $maxsid(P)$ plot, we can see that no one simulation dominates the others in terms of maximum speed. From the $C_b\_of\_max\_speed(P)$ plot, we can draw modest conclusions, but it would be difficult to claim that this term is greatly affecting which simulations have the maximum speed. From the $C_{tv}\_of\_max\_speed(P)$ plot, we can see that most of the high speeds either come from very low or very high turbulent viscosity coefficients (colored blue and red, respectively). We quantified this observation (see table 5.6), and found that the simulations with extreme turbulent viscosity coefficients had over three quarters of the total area, meaning that the relationship between high speeds and turbulent viscosity is large.

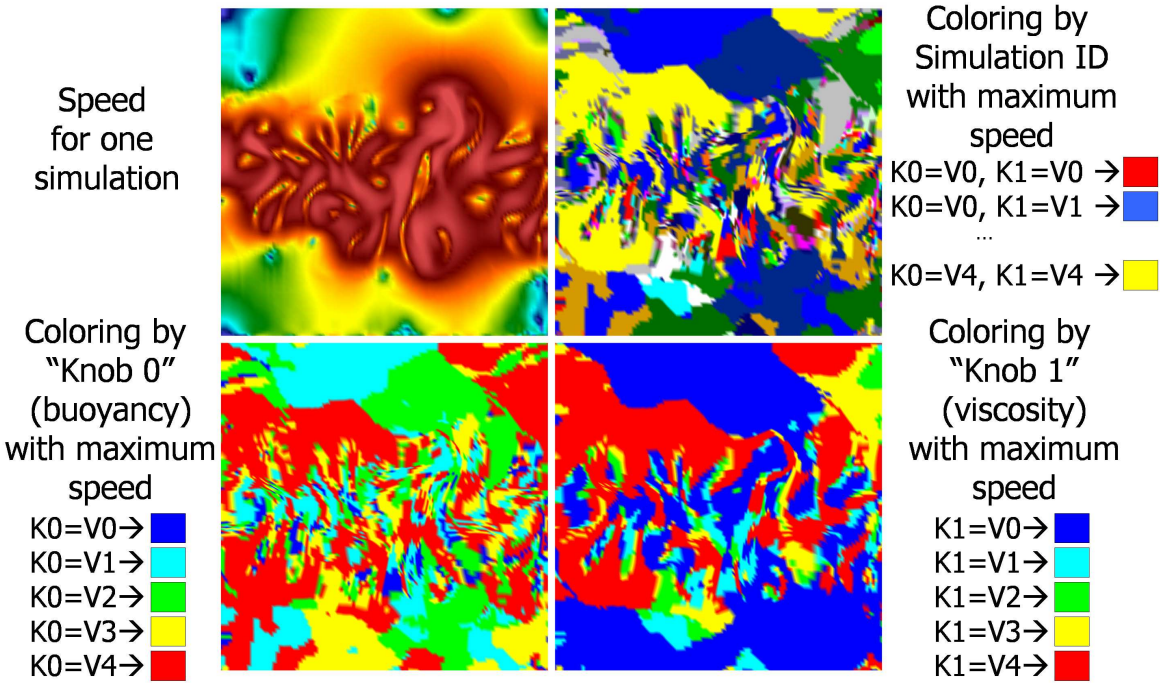

Figure 5.7: In the upper left, we see a normal rendering of speed for a single simulation. In the upper right, we color by $maxsid(P)$. In the lower left, we color by $C_b\_of\_max\_speed(P)$. In the lower right, we color by $C_{tv}\_of\_max\_speed(P)$.

Coefficient	very low	low	middle	high	very high
Buoyancy	4.2%	21.1%	20.4%	17.5%	36.8%
Turbulent Viscosity	47.2%	8.1%	4.8%	8.7%	31.0%

Table 5.6: Quantifying how much space each coefficient covered in terms of percentage of the total space.

Finally, we were also interested in quantifying the changes from simulation to simulation in our parameter study. We did this by calculating the following derived quantities:

1. $\frac{\sum_{all\ 25} speed}{25}$
2. $\max_{all\ 25} speed$
3. $\min_{all\ 25} speed$
4. $\max_{all\ 25} speed - \min_{all\ 25} speed$

The results are shown in figure 5.8. The fourth quantity informs an analyst as to the maximum differences possible for each point in space. Since we are performing

these operations on a parameter study, we are effectively quantifying the uncertainty for this simulation. Of course, there are many alternative ways that an analyst may want to construct uncertainty information from ensembles of simulations. But we believe that through the examples we have presented in this section, we have motivated the capability of our system to do so and the importance of a flexible and powerful derived quantity system.

5.6 Summary

We have demonstrated a powerful system for data-level comparative visualization and analysis. The system we have developed supports a wide range of comparative analyses including comparisons in physical space, in logical space, across symmetry conditions, with analytic functions and between simulated and experimental results. In addition, the system is very versatile not only in the modalities of cross-mesh field evaluation it offers, but also in the range of derived quantities that can be generated. Finally, it supports these operations in a highly scalable, distributed memory, parallel computational paradigm, and we believe this is the first time this issue has been discussed in the context of visualization.

We demonstrated some powerful applications of this system. The As-Built Modeling section demonstrated the need for parallelization. The volume overlap calculations in section 5.5.1 and analysis of a parameter study in section 5.5.2 motivated the power of combining data-level comparisons with a richly featured analysis tool. Later in section 5.5.2, we presented a new, novel technique for visualizing time-varying data by focusing on binary conditions and leveraging comparative techniques. Lastly in section 5.5.2, we presented another new, novel technique for visualizing large ensembles of simulations. In addition, we demonstrated how a system such as ours can be used to calculate uncertainties and postulate interesting questions for examining ensembles.

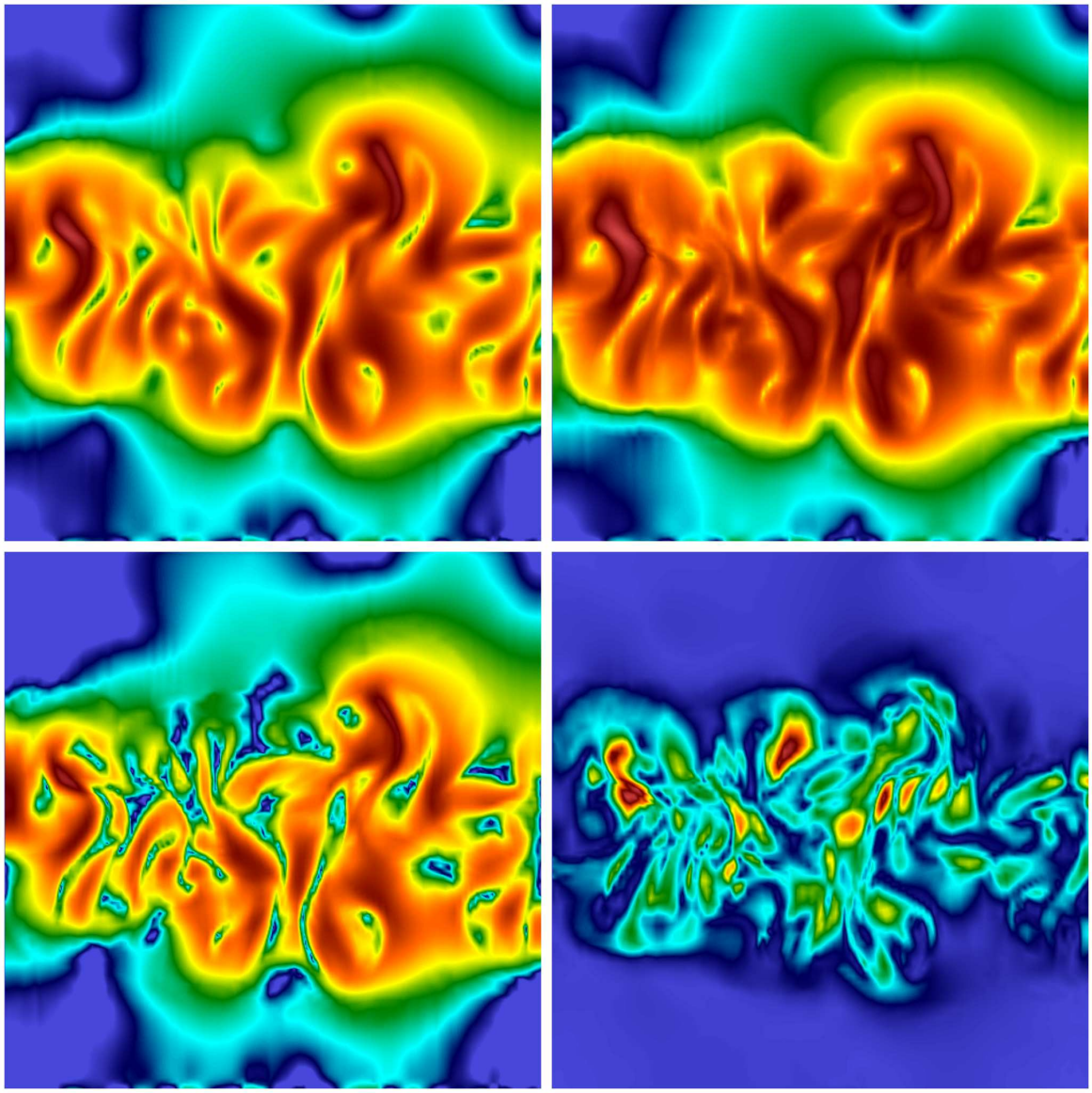

Figure 5.8: In the upper left, we are coloring by average speed, in the upper right by maximum speed, in the lower left by minimum speed, and in the lower right maximum variation. The first three plots vary in speed from zero to two and are colored using a logarithmic scale. The fourth plot (of differences) ranges from zero to one and is colored using a linear scale. These plots effectively quantify the uncertainty for this ensemble of simulations.

Part II

Improving the legibility of large data

Chapter 6

Using Basic Quantitative Techniques to Improve Legibility

Subsequent chapters discuss advanced techniques for improving legibility. In this chapter[1], we focus on more basic techniques for reducing data set complexity, most of which are already well-known. These basic techniques can be characterized in two distinct ways:

1. process some of the data; a specific region of interest
2. process all of the data set, but simplify it somehow

Of course, these two methods can be combined as well.

The first method, process some of the data, is important when an end user wants to explore what is happening in a specific region. Examples range from examining an area where two materials collide in an engineering simulation to looking at an area of mixing in a turbulence calculation. The first two sections, Culling By Reference (6.1) and Culling By Value (6.2), describe techniques that allow the user to cull data and focus on a specific region of interest.

The second method, simplifying the data set, is described in the Quantitative Analysis section (6.3), which deals with operations that dramatically reduce the scale of the data.

[1]Much of the text from this chapter comes from [19], which was a collaboration between Hank Childs and Mark Miller

The principal, new contributions of this chapter are:

1. The idea of using a Subset Inclusion Lattice data structure as the basic descriptor inside a visualization and analysis tool for characterizing complex relationships.

2. The notion that the assets required by visualization tools are very similar to the assets required by analysis tools, hence a combined visualization and analysis tool is cost efficient.

3. Further motivation that interoperable features within a richly featured tool can lead to powerful analysis.

6.1 Culling By Reference

Simulations often decompose their data in a variety of ways: by files, by materials, by parts in an assembly, by processor pieces, by levels and patches in an AMR hierarchy, etc. In addition, there are often key subsets in the data, such as nodes and/or zones representing boundary conditions, slide surfaces, user-defined tracers and probes.

We call the technique for managing complexity described in this section *Culling By Reference*. The technique enables the user to select which parts of the data set are culled and which are displayed using pre-defined subsets of their data (like the examples described in the paragraph above). We use a concept from the Sets and Fields (SAF) library[52] called a Subset Inclusion Lattice (or SIL). The SIL incorporates concepts from graph theory and set theory and enables users, for example, to turn on and off certain materials or processor pieces in the display, or to vary the AMR resolution at which data is being displayed.

The SIL is in an encoding of subset relations in graph form. Strictly speaking, a SIL is a directed, acyclic graph of the inclusion (subset-of) relationships of partially ordered sets representing the join-irreducible infinite point-sets of some computational mesh (see [12]). While the SIL necessary to represent relationships for a visualization and analysis tool does not support this strict mathematical interpretation, it is similar in concept and is nonetheless called a SIL, although it might be more appropriate named a lite-SIL.

Now let's discuss the form of our SIL in more detail. The SIL encodes relations

between the sets (such as subset relations), as well as the corresponding category of a set (for example the aluminum set is a subset of the whole set using the material category). The SIL is a graph made up of two sets of nodes: one set corresponds to categories placed on the data, the other corresponds to subsets of the whole data set. Edges in the graph always are incident to both sets - describing relationships between subsets and defining the category of subset. Further, this makes our graph bipartite.

Now consider a more concrete example of this graph (see figure 6.1). One node of the graph corresponds to the whole data set and it is referred to as the whole. **m**aterials and **p**rocessors, two category nodes, have directed edges to **w**hole. Under **p**rocessors, there are many subset nodes, **P**roc X, each one corresponding to the subset of the data set from processor X. More subset nodes - **p**lastic, **a**ir, and **a**luminum - have directed edges to **m**aterials.

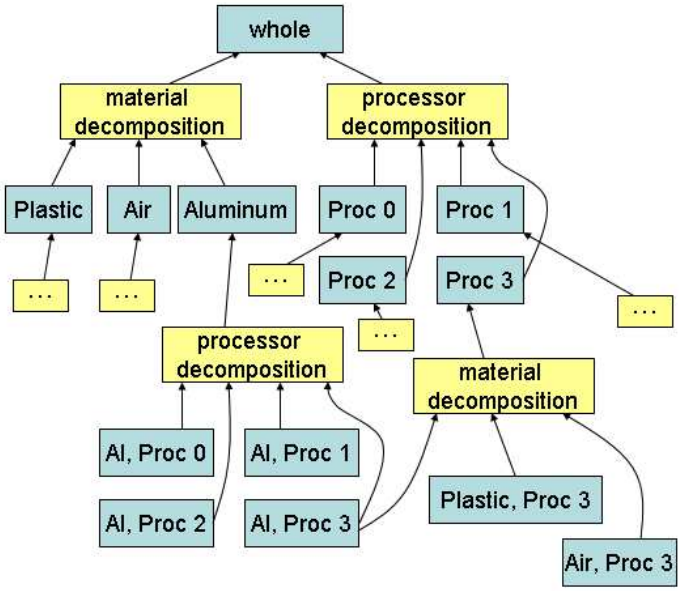

Figure 6.1: An example graph of the data set. Subset nodes are colored blue, category nodes are colored yellow.

The SIL is powerful because it is a general framework for representing diverse data in an intuitive way. It provides ultimate flexibility in creating multiple subset hierarchies of a data set. The SIL may have an arbitrary number of subset and category nodes and may be arbitrarily deep. In addition, it is possible to construct GUI controls to manipulate any type of SIL (see figure 6.2), even if its form in not known a priori. This enables users, for

example, to turn on and off familiar subsets of their data that are specific to their simulation, such as certain materials or domains, or to vary the AMR resolution at which data is being displayed. In addition, the SIL provides valuable controls to the user to throttle the amount of complexity being displayed in any one view.

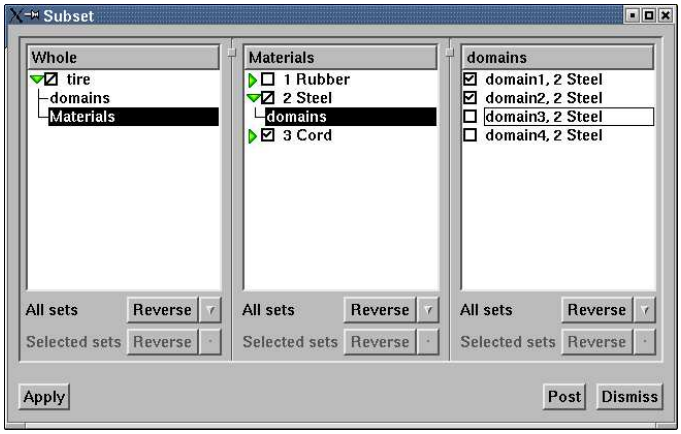

Figure 6.2: The subset selection window allows users to choose which subsets of the data set they would like to view.

6.2 Culling By Value

In the previous subsection, we described a data culling technique based upon selecting from among pre-defined subsets. In this section, we describe another culling technique based upon data value which we call *Culling By Value*. There are two general techniques for doing this: to eliminate portions based on data values or to eliminate portions based on spatial position. It should be noted that the techniques presented in this section are well understood and are being presented for completeness.

We have found the most powerful operation to cull by value is isovoluming. Isovoluming allows a user to specify a range of interest for a specific scalar quantity and then remove all portions of the data set outside that range (see figure 6.3). For example, a user could restrict the data set to temperatures between forty and fifty degrees Celsius. Of course, some elements may have a portion inside the range and a portion outside the range. In this case, the isovolume operation will subdivide the element (into tetrahedra) so that the remaining portion is entirely within the range. In addition, the isovolume operation

can be applied multiple times with different variables, providing users the ability to ask questions like: "what portion of the data set has pressure between P1 and P2 and density between D1 and D2?" [74] contains more motivation for these types of sub-selections.

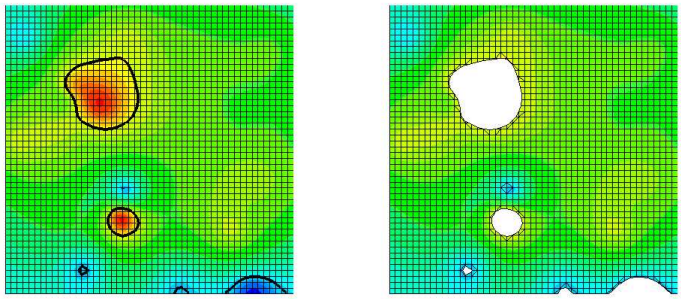

Figure 6.3: On the left, a data set is shown with thick black lines indicating the areas outside the isovolume's range. On the right, the data set resulting from the isovolume operation is shown.

The ability to make new derived quantities enhances the utility of isovoluming. Derived quantities can transform non-scalar data into scalar data, for example the magnitude of a vector or the maximum shear component of a tensor, which can then be used with the isovolume operation. In addition, the derived quantities can be used to construct new quantities that give greater insight into the data set. A good example comes in [34], which outlines the $\lambda 2$-method. This method creates a new derived quantity which identifies vortical flows. The method involves taking the gradient of the velocity field, separating out and then re-combining the symmetric and anti-symmetric portions of the resulting tensors, and solving for its eigenvalues. Vortices can then be identified as connected regions where two of the eigenvalues are negative. These regions can be easily found using the isovolume operation.

When culling by value is applied to coordinate fields, the effect is to cull portions of the data set by spatial position. Many of these techniques are well known: slicing by a plane, slicing by a parameterized surface, clipping by a plane to produce a subvolume, restricting the data set to a box, etc. These techniques are simply enumerated here for completeness. However, the next section, using quantitative techniques, will leverage these operations, as well as the isovoluming technique described above.

6.3 Quantitative Analysis

Queries are sinks in a data flow network that produce summary information about a data set. We distinguish between two classes of *queries*. The first class, called *point queries*, returns information about a single element. Examples of this include the ability to query the variable values of a specific element, or to perform this operation for all time states. The second, more powerful class, called *aggregate queries*, calculates information about an entire volume or surface. Examples of this include the ability to calculate centroids or moments of inertia of a volume, integrate a quantity over a volume, calculate the flux along a surface, find the minimum and maximum values of a quantity, etc. Of course, the exact functionality within the direct and aggregate queries required to perform meaningful analysis is dependent on the type of simulation being studied.

One of the most compelling aspects of the architecture presented in this dissertation is the way that queries can be combined with other features. For example, an end user can use the SIL mechanism to limit the data set processed to a single material of interest. The user can then remove all portions of that material on one side of a plane. Then the user can create a new derived quantity that multiplies each element's density with its volume, to give the mass per element. Then an aggregate query can be applied to give the total mass in the volume on the desired side of the plane limited to that material. And this query can be computed as a function of time. This, of course, is just one example of the way that features can be combined. But this example motivates the utility of a richly featured application that has strong quantitative capabilities.

In the example above, we focused on specific data manipulations (isovoluming and clipping), derived quantity calculations (mass), and queries (integration). But it is often necessary to add new capabilities in any of these areas. So it is important to keep in mind that data flow networks provide a highly extensible and customizable framework and that this framework allows for new quantitative features, new data manipulations, and new derived quantities to be added and combined seamlessly. This extensibility is crucial for providing custom analysis for different types of simulations.

More generally, consider how queries fit into data flow networks. Data flow net-

works typically consist of a source (i.e. file reader), filters, and a sink (i.e. rendering module). But, with queries, we are introducing a new genre of sinks that produce numbers instead of pictures. More importantly, queries are extremely effective at reducing complexity and/or improving legibility. Also, these queries can be combined with the assets of visualization data flow network implementations - file readers and data manipulation - with all of the associated interoperability.

Finally, this infrastructure is all the more powerful when it is combined with a scripting language, as has been done with VisIt with its Python-based[56] command line interface. This is especially important because it allows users to put together automated "report cards" that describe their data sets. For example, a user could combine the $\lambda 2$-method described in Section 6.2 with the quantitative techniques in this section to determine the amount (i.e. volume) of a simulation undergoing vortical flow. This one number characterizes the data set (and reduces its complexity), allowing it to be compared to other simulations. So these report cards allow for comparison of related simulations. Further, they can be used to compare the thousands of related simulations that occur when doing parameter studies.

Chapter 7

A Statistical Approach for Improving Legibility

In this chapter[1], we introduce *Equivalence Class Functions (ECFs)*. ECFs are a new class of derived quantities designed for visualization at the extreme scale. These functions are defined over equivalence classes of elements from the original mesh obtained from basic groupings. When used directly, ECFs offer more manageable and meaningful visualizations. Alternatively, ECFs can be used to synthesize new fields on the original mesh to help reveal regions of importance.

ECFs offer a new, more powerful kind of data culling based on classifying regions of mesh according to statistical relationships in fields defined on the mesh. They can be used in the visualization process in two fundamentally different ways. First, an ECF can be created and then visualized directly with traditional techniques – using the ECF in an *analysis*-oriented manner. Second, an ECF can be used to synthesize new derived quantities on the original mesh – using it in a *synthesis*-oriented manner. When used for analysis, ECFs can reduce data from the extreme scale to something that is more manageable and meaningful, thereby providing more informative quantities to visualize. When used for synthesis, ECFs

[1]Much of the text from this chapter comes from an upcoming submission to the journal IEEE Transactions on Visualization and Computer Graphics. The original idea for this chapter, initial version of a paper, and implementation were by Hank Childs. However, Mark Miller took over first authorship of the upcoming submission and it is important to note that his editing of Hank's initial draft, which was substantial, is what appears in this dissertation. Further, Dr. Miller, along with Ken Joy, contributed significant ideas, including renaming the concept from derived data functions to equivalence class functions.

can be used to identify regions in the original mesh deemed *important* because, for example, statistical behavior is significantly different than the average for elements in the same class. Applied in this way, ECFs help to quickly and effectively reveal the salient portions of complex data sets.

ECFs are differentiated from previous derived quantity efforts by their ability to survey and summarize the whole data set, employ statistical reductions, and the ultimate synthesis of this information back onto the original mesh. After describing the basic technique, we demonstrate results from the application of ECFs to the visualization and analysis of turbulent mixing and uncertainty from equations of state.

7.1 Related Work

Several previous publications have presented the idea of creating derived quantities from the original data set and visualizing those quantities. Many richly featured visualization systems, such as OpenDX[1], EnSight[21] and SCIRun[35], provide subsystems for generating derived quantities. Clyne and Rast discussed the analysis and visualization of quantities derived from high-resolution numerical turbulence simulations outputs, exploring the error introduced when computing new data fields from wavelet coarsened approximations of the original data [20]. The systems described in these papers are very powerful. But they are often limited to generating only fields that are defined on the original mesh. Unlike ECFs, they lack the ability to survey and summarize the entire data set onto a new mesh. However, with respect to creating derived quantities on the original mesh, two fundamental traits are shared between these systems and our implementation of ECFs. First, each has a rich set of operators with which to create new fields. For example, there are operators for arithmetic sum and difference, for the vector derivatives, gradient, Laplacian and curl, for logical if-then-else and many, many more. The second is the ability to combine these operators in arbitrary ways. This is critical to leveraging the true power of such a system.

Within the visualization community, ECFs are related to bin smoothed scatter plots. While a number of papers discuss the application of scatter plots to information visualization, a fewer number of papers, including [8] and [81], discuss their application to

scientific visualization problems. In fact, most of the latter applications center around flow visualization problems using linked views and/or brushing techniques, [8, 30, 36]. When the ECF summarization operator is *average*, an ECF will indeed result in a bin-smoothed scatter plot also known as a *regressogram*. But the similarity to these well-known techniques ends there. ECFs admit a wide variety of summarization operations. Furthermore, all the previous work related to ECFs is focused on what we call *analysis*-oriented use, while we also support *synthesis*-oriented use of ECFs. Finally, no one has discussed the integration of the kinds of statistical analyses ECFs permit with turnkey, richly-featured, highly scalable visualization tools.

Outside the visualization community, ECFs are related to several concepts. Quantization is the concept of replacing a continuous quantity with a finite set of discrete values, [83]. The goal of quantization is usually data reduction and approximation of the original set of values. ECFs are a broader concept that includes not only approximation but also arbitrary comparison. Some quantization concepts may be useful in constructing equivalence classes, [89]. Categorical data analysis, [2], developed mostly in the 1960s, concerns the dependence analysis of "individuals" that fall into categories over several attributes. Individual mesh elements would be the data set "individuals" for a categorical analysis. This analysis begins by counting the individuals in a multidimensional table defined by the available variables that have been quantized. Some useful concepts from this domain include marginal tables obtained by collapsing over some dimensions and conditional tables obtained by taking a particular slice of the table. Statistical assumptions used in analyzing such tables involve multidimensional probability density functions (PDF) and consequently marginal PDFs and conditional PDFs. Various methods exist for estimating such PDFs from the multidimensional table or from the original data [68]. The computation of ECFs described in this chapter is useful for extending analysis of categorical data to larger data sets.

7.2 Concept

In this section, we will formally define an ECF, discuss how one is constructed, and describe the utility of various summarization operators. In the succeeding section, we will discuss the details of our implementation.

An equivalence class function is constructed using basic statistical analyses to group the collection of elements from a computational mesh into equivalence classes. A summary value is computed for each equivalence class by applying a *summarization* operator to all the elements in each class. The result is a function defined over a uniform or rectilinear mesh of equivalence classes. The diagram in figure 7.1 illustrates these concepts. The remainder of this section formalizes them.

7.2.1 Interpreting Elements of a Mesh as Points in a High Dimensional State-Space

In scientific simulations, a problem to be simulated is discretized into a number of tiny pieces called *elements*. Scalar, vector and/or tensor fields are defined over these elements. Examples are pressure, velocity, and stress. These fields need not represent physical properties and may describe literally any quantity the simulation developers choose such as processor rank, amount of damage to a material, counts of events and so forth. The collection of all such fields are the *dependent variables* of the simulation. The dependent variables define the *range* of a mapping from a *domain* defined by another set of variables called the *independent variables*. Typically, the independent variables are simply the spatial coordinate fields. Lastly, we refer to the set made up of the union of dependent and independent variables as the simulation's *state variables*.

Each state variable defined on an element can, in general, vary over the extent of the element. Nonetheless, for the purposes of simplifying the discussion, we will assume every state variable is constant valued over each element it is defined on. In other words, our discussion will focus on the case of piecewise-constant or *element-centered* fields. In a later section, we explain how the construction described here can be extended to the more general case. Hence, we can construct a tuple of state variable values for each element. We

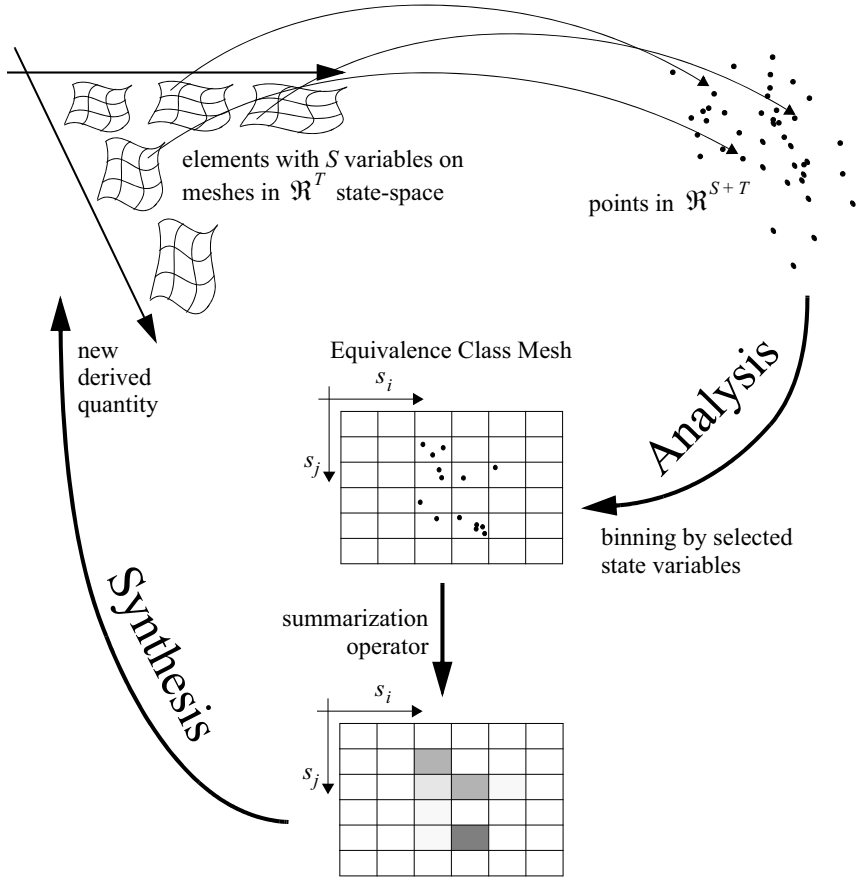

Figure 7.1: Conceptual Diagram of ECF Generation. Elements from the input mesh(es) (upper left) are mapped into a high dimensional space (upper right) defined by the simulation's state variables. This cloud of elements is partitioned into equivalence classes (the analysis step). Summary values are computed for each equivalence class (the summarization step). The resulting mesh and field can be visualized directly or can be used to synthesize a new variable onto the original mesh(es) (the synthesis step).

call this tuple the *element state tuple*. For example, a tuple such as $\{x, y, rho, p\}$ might represent the element state tuple for elements from a simple, 2D mesh in the xy-plane upon which two dependent fields, density, rho, and pressure, p, are defined, and x and y are fields corresponding to the centroid of the element.

Finally, note that any derived quantities that are defined on the original mesh can be treated in a completely analogous manner. They too, can be viewed simply as additional entries in an element's state tuple. This allows for derived quantities, including those which are synthesized from an ECF, to be included in other ECFs.

7.2.2 Incorporating State Space Variables

If there are S state variables, each element's state tuple is an S-tuple[2]. If there are N elements in the mesh, there are N S-tuples to denote a single instance of the state of the simulation. More typically, however, there is not just a single state instance. Instead, there is often a set of states that are related to each other. For example, there is often a time series. Or, in the case of parameter studies, there is a time series for a set of parameter choices.

We can accommodate these additional degrees of freedom, folding them into our current construction, by appending terms to the independent variables and expanding our element state tuples. If there are T additional independent variables defining M different states in the state space, then over all states of the simulation, we have MN tuples each of size $(S+T)$.

The MN, $(S+T)$-tuples from a simulation will now be denoted as n data *vectors*, $\mathbf{q}$, of dimension $d = (S+T)$. We also let $\mathbf{Q}^{\min}$ and $\mathbf{Q}^{\max}$ be the data vectors formed by taking the component-wise minimum and maximum values over all n. So, over all states and over all the elements in each state, the original simulation data takes the form of a cloud of points, the *element cloud*, in $\Re^d$ bounded by the d-dimensional hyperbox with corners $\mathbf{Q}^{\min}$ and $\mathbf{Q}^{\max}$. For convenience in notation, we shall index the components of $\mathbf{q}$ (e.g. the state variables) by the symbol s_i such that $\mathbf{q} = [q_{s_1}\ q_{s_2}\ q_{s_3}\ \ldots\ q_{s_d}]$. Finally, we let $Q = \{\mathbf{q}_i \text{ for } i = 1 \ldots n\}$.

7.2.3 Equivalence Relations, Subsetting and Data Classification

An equivalence relation, [6], partitions a set into a collection of disjoint subsets. The subsets are called *equivalence classes*. Because the set Q represents points in the *state space* of a simulation, equivalence relations on Q permit us to utilize state variables in a myriad of interesting ways to partition Q into subsets that are highly relevant to the underlying analyses.

[2]We can accommodate cases where state variables are defined on only some of the elements by using a suitable value, such as NaN, for the remaining elements, to indicate the associated state variable is not applicable to them.

Put another way, treating the elements of meshes from scientific simulations as a set of points in a high dimensional state-space and then applying equivalence relations to partition that set creates a new and powerful tool to classify and cull regions of a mesh based upon statistical relationships between the quantities defined upon the mesh.

7.3 The Equivalence Class Mesh

We begin the construction of the underlying mesh for an ECF, the *Equivalence Class Mesh (ECM)*, by selecting a subset of the d axes of $\Re^d$ and partitioning each into intervals or *bins* in a manner analogous to a histogram. We let D' be the set of selected state variable indices, i. Thus, state variable s_i is selected iff $i \in D'$. Also, denote $d' = \text{rank}(D')$, the number of variables in s_i selected.

The set of selected state variables are used to construct the *domain* of the ECF. For an arbitrary state variable, s_i, partitioning results in a set of intervals defined by

$$I_{s_i}^k = \left[Q_{s_i}^{\min} + H_{s_i}(k), Q_{s_i}^{\min} + H_{s_i}(k+1)\right) \text{ for } k = 0 \ \ldots \ N_{s_i} - 1 \tag{7.1}$$

where

$$H_{s_i}(k) = \begin{cases} kh_{s_i} & \text{for uniform binning} \\ \\ \displaystyle\sum_{j=0}^{j=k} h_{s_i}(j) & \text{for non-uniform binning} \end{cases}$$

For uniform binning,

$$h_{s_i} = \frac{(Q_{s_i}^{\max} - Q_{s_i}^{\min})}{N_{s_i}}$$

and N_{s_i} is the number of desired bins for state variable s_i. In this simplest of cases, the number of bins, N_{s_i}, for each state variable s_i are the only user specified parameters. More advanced alternatives have been considered, but are not presented in this document.

Thus, for each of the d' state variables the user selects, we obtain a set of N_{s_i} intervals,

$$I_{s_i} \ = \ \{I_{s_i}^k \text{ for } k = 0 \ \ldots \ N_{s_i} - 1\} \tag{7.2}$$

The equivalence class mesh for the selected state variables is then defined by the Cartesian product of these sets of intervals

$$\mathbf{M}_{D'} = \prod_{i \in D'} I_{s_i} \tag{7.3}$$

which for uniform binning generates a uniform mesh and otherwise generates a rectilinear mesh. Note that the members of $\mathbf{M}_{D'}$ represent d'-dimensional hyperboxes or *bins* in $\Re^{d'}$ and can be logically arranged in an array such that

$$\mathbf{M}_{D'}(j1, j2, \ldots, jd') = I_{s_{i1}}^{j1} \times I_{s_{i2}}^{j2} \times \ldots \times I_{s_{id'}}^{jd'} \tag{7.4}$$

An arbitrary point, $\mathbf{q}$, in the element cloud Q lands in bin

$$\mathbf{M}_{D'}(j1, j2, \ldots, jd') \text{ iff } q_{s_i} \in I_{s_{i1}}^{j1} \; \forall \; i \; \in \; D' \tag{7.5}$$

7.4 Summarization Operators

The ECM partitions the element cloud into a uniform or rectilinear arrangement of equivalence classes (or bins). Each class holds a variable number of points. Some may be empty. So, we have a set-valued, piecewise-constant function. The *value* of this function at a given point in the ECM is the associated equivalence class. Note that this intermediate representation of the data, as a set-valued function, is primarily a conceptual tool for describing ECFs and would most likely never be instantiated in any actual implementation.

We seek a way of transforming this rather complicated function into one that can be easily visualized directly using traditional techniques (e.g. a simple scalar, vector or tensor field) and/or can be used to synthesize a new field on the original mesh. In other words, we would like to find some way of taking the set of points in an ECM bin and computing a single scalar, vector or tensor quantity. This involves applying a *summarization* operator to the set of points in each equivalence class and computing a single, summary value for each class.

In typical use, a user would select one of the state variables, different from those used to construct the domain of the ECF, to serve as the ECF *range*. In theory, however,

there is no reason to limit a summarization operator to acting only on one state variable. Summarization could involve multiple state variables.

The possibilities for candidate summarization operators are almost limitless and vary significantly depending on the analyst's needs. Some examples of the possibilities are briefly described below.

7.4.1 Summarization by Averaging

A common use case is to select one component of $\mathbf{q}$ (e.g. one state variable) and then average this value over all members of each equivalence class. If the domain of the ECF is defined using the original mesh's independent variables, then this form of summarization yields a reduced resolution representation of the original mesh. On the other hand, if the domain of the ECF is something other than the original mesh's independent variables, averaging as a form of summarization yields a *bin-smoothed scatter plot*.

7.4.2 Summarization by Weighted Averaging

We can also compute summary values in a way which emphasizes some members in a class over others. For example, we can perform a volume weighted average, or if we are performing an analysis over time, we can weight elements based on distance in time from an important event.

7.4.3 Summarization by Class Rank

We can also use the *rank* of an equivalence class (e.g. the number of members in that class) as a summarization operator. The resultant ECF has a number of interpretations relevant from a statistical standpoint. In the case of a one dimensional domain, this form of summarization yields a *histogram*. In multiple dimensions, the resulting ECF can be used to compute non-parametric estimates of the *joint probability density functions* of the selected state variables.

7.4.4 Summarization of Vector and Higher Rank Quantities

The summarization process need not result in a single scalar value for each class. If we are summarizing vector data, for example, we can compute a summary *vector* for each class. In some cases, a simple vector sum may be appropriate. For example, if we are summarizing a displacement vector, we may be interested in whether or not displacements by all the members in a class tended to cancel or coincide with each other. In other cases, we may be more interested in direction of the summary vector and not so much in magnitude. For example, in the case of a flow field, we may be interested in how much the direction of flow varies over the members of a class, independent of variations in magnitude of the flow.

7.5 Applications

A Richtmeyer-Meshkov (RM) calculation simulates the mixing of heavy and light gases as a shockwave passes through them. The canonical orientation is one in which the shock wave propagates in the positive X-direction passing first through the light gas to the left and then through the heavier gas to the right. Typically, analysts study the pressure over the mesh. The figures below demonstrate the application of standard techniques to visualize pressure from a 25M element RM calculation from LLNL's RAPTOR simulation code. Figure 7.2 shows an xy-slice through the mesh while figure 7.3 shows a set of iso-surfaces. Although some variation in pressure is evident in the slice, it does not appear to vary significantly. Furthermore, the iso-surfaces tend to appear as parallel sheets with small deformations. A lot of detail of turbulent mixing in the data is not evident by applying standard techniques to the native quantities.

Equivalence class functions can be used to more effectively visualize this problem. First, all of the elements with similar X-values can be grouped together to form an equivalence class. Second, the functions "average" and "standard deviation" can be defined over the equivalence classes. Simply plotting these 1D functions is informative (see figure 7.4). However, the real discovery in terms of how local trends deviate from global trends is achieved when synthesizing these functions back into the original data set (see figures 7.5 and 7.6).

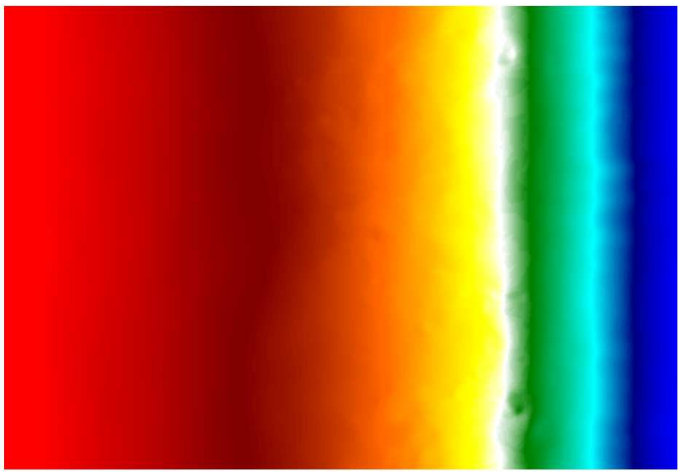

Figure 7.2: A pseudocolor of pressure along an XY-slice. For pressure, the global trend dominates local variations.

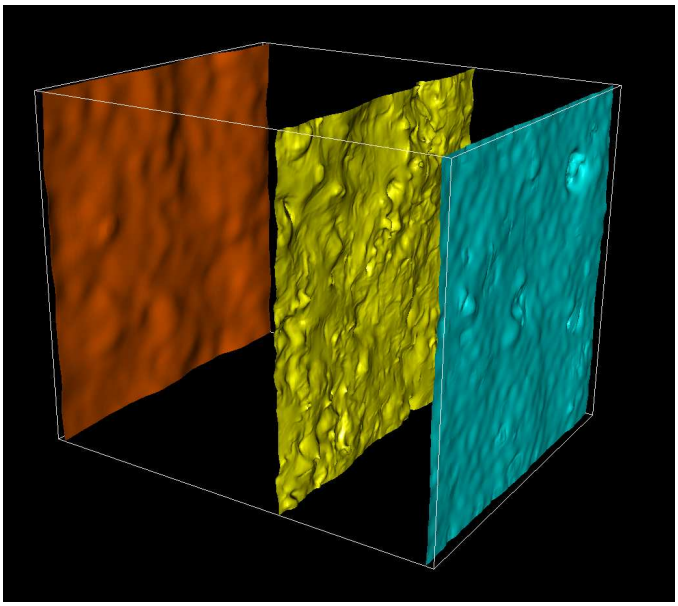

Figure 7.3: Contours of pressure. This results in essentially axis-aligned planes, because, again, global trends are dominating local trends.

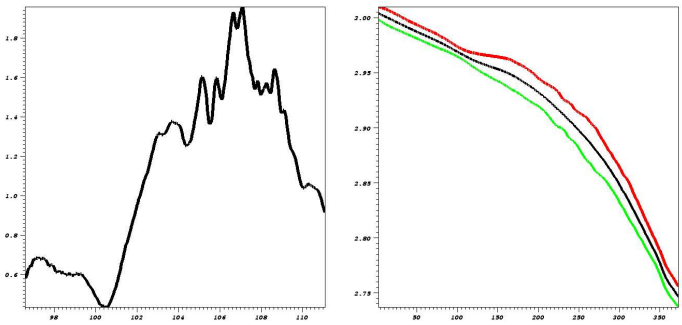

Figure 7.4: On the left, we plot standard deviation as a function of X. This is obtained by grouping all elements with similar X-values into equivalence classes. On the right, we plot average pressure as a fuction of X (black), average plus ten standard deviations (red), and average minus ten standard deviations (green).

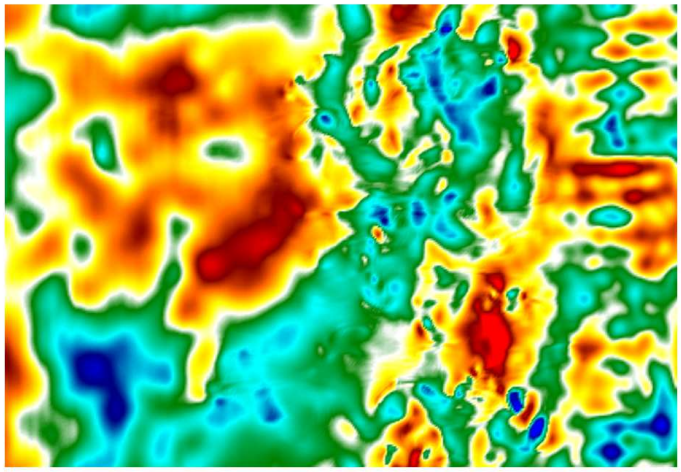

Figure 7.5: An XY-slice of the number of standard deviations each point is away from the global trends. (This is (pressure - ECF_A) / ECF_B, where ECF_A is an ECF of average pressure based on X groupings and ECF_B is an ECF of the standard deviations of pressure over the X groupings. With this picture, for the first time, we can see deviations from the global trend.

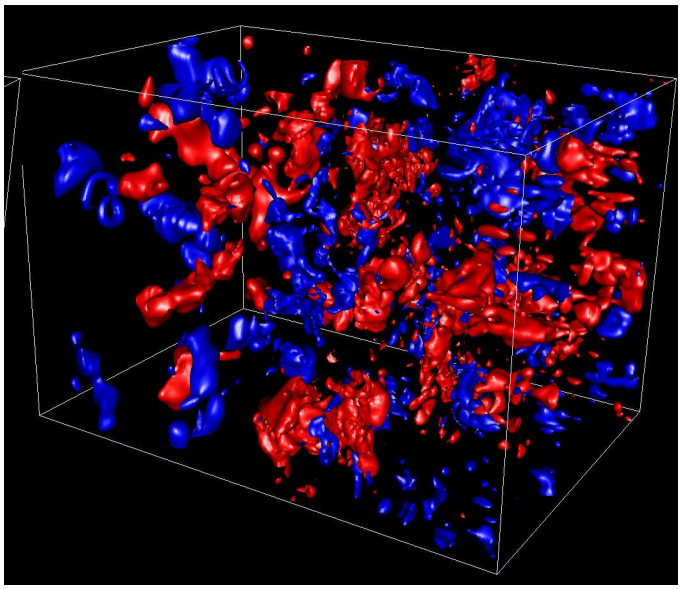

Figure 7.6: Isocontouring the synthesized ECF. Red regions are two standard deviations above the global trend, blue are two standard deviations below.

Chapter 8

Line Scan Techniques for Shape Characterization

Too often, analysts compare two shapes by placing their pictures side-by-side and declaring them "close enough". When presenting results to colleagues, these analysts will overlay two viewgraphs of the two shapes on an overhead projector so that the audience can see where the shapes agree and disagree. This practice has led to the phrase "viewgraph norm", where the analyst makes an "expert judgment" and declares that two shapes are either similar or dissimilar.

In this chapter, we consider shape characterization functions. These functions serve as both a way to describe a single shape and also as a way to quantitatively compare shapes that goes beyond the "viewgraph norm". These functions are examples of the types of analytics that fit well within the architecture we have proposed in this dissertation.

We discuss six shape characterization metrics:

1. Chord length distribution (individual)
2. Chord length distribution (aggregate)
3. Ray length distribution (individual)
4. Ray length distribution (aggregate)
5. Volume as a function of length scale

6. Mass as a function of length scale

8.1 Definitions

8.1.1 Chord and Ray Length Distributions

A *chord length distribution* is a probability distribution over lengths. If a line intersects a shape Q to form a chord, the chord length distribution for that shape, $\theta_C(l)$ will tell you the probability that that chord will be a certain length, l.

A *ray length distribution* is a probability distribution over all rays inside a shape Q. If you place a random ray (emmanating from a random point in Q, and going a random direction), then the ray length distribution for that shape, $\theta_R(l)$, will tell you the probability that that ray will go a certain length before exiting the shape.

Non-convex shapes create an ambiguity in the definitions above. The intersection between a line and a shape can form multiple, disconnected segments. There are two options for analyzing the resulting segments:

- The segments can be considered together, that is the length is the sum of the lengths of each segment
- The segments can be considered separately, that is each segment independently contributes to the probability distribution.

The first variant is referred to as *aggregate*, while the second variant is referred to as *individual*.

Finally, these distributions accept weights, which are typically the density. With weights, the probability distributions are not over lengths, but actually over "weighted length", which can be referred to as "mass" or "linear mass".

The aggregate version of the chord length distribution has been well studied [57, 47, 49, 48]. [57] also contains results related to the individual version of the chord length distribution.

8.1.2 Volume and Mass as a Function of Length Scale

It is clear that determining how much mass (or volume) is at a given length scale is a pertinent metric. However, in two- and three-dimensional space, length scale is somewhat ambiguous. So defining this metric requires defining what it means to be at a given length scale. Here are some possible definitions:

1. A single length for an object,
2. A field varying over the object, where each point has its own length,
3. A field varying over the object and over angle, where each (point, angle) pair has a length.

Option #1 is probably incorrect, since there may be great deviation in the object. By grouping the entire object into just one length scale, there is a loss of information.

Option #2 is not well-defined. For a given point in the object, what should the length be? Should it be the length of the minimum chord that goes through it? The maximum chord? The average of all chords? All the aforementioned definitions can skew results. Consider figure 8.1. Using the minimum chord that goes through a point would give short length scales to the upper corners and short valleys of the cyan object, even though intuition would say they are at longer length scales. Alternatively, using the maximum chord that goes through point P would yield a large length scale, again where intuition would say that the length scales for the points in the jut that contains P should all be small. Finally, the average of all chords is also a poor choice. If you average all chord lengths surrounding P, the resulting length, L_0, is not even a possible chord length for P. Restated, there does not exist an angle α such that the chord through P at angle α has length L_0. So it is difficult to choose a single length scale for a point and have it meet our intuition.

Option #3, on the other hand, is well-defined. For a given point and a given angle, the length is simply the length of the chord through the object. (Note that, technically, the point and angle define a line, and the intersection of that line will result in one or more chords. In this case, the length is the length of the single chord that contains P.) Note

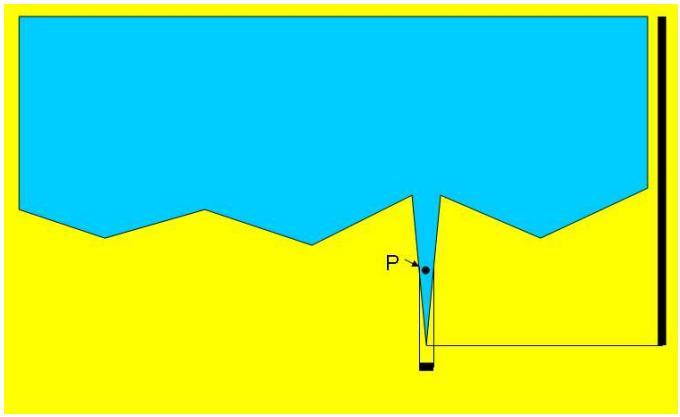

Figure 8.1: Choosing a single value as the length scale for P is difficult. The length could be as small as the short, horizontal segment, or as long as the long, vertical segment.

that this approach (having many length scales stemming from a single point), mirrors the interpretation of individual chord length distributions.

Assuming the definition of length scale above (option #3), we can then define formulas for:

1. Volume below some scale α,
2. Mass below some scale α,
3. Distribution of volume by scale,
4. Distribution of mass by scale.

Note that these quantities are highly related. If we can calculate the distribution of mass by scale, we can calculate the distribution of volume by scale by using a uniform density. Similarly, if we have a distribution of mass by scale, we can use that distribution to directly calculate the total mass below some scale α. However, for ease of presentation, we present the simplest metrics first and then build up. We start with a definition for how much volume is at a scale below some α. We then create a similart definition for mass by adding a density function. Finally, we give definitions of volume and mass as distributions.

First, however, we will define utility functions that help with our definitions. These definitions suggest seemingly computationally expensive solutions. In section 8.2 and later in appendix B, we will show how a probabilistic, line scan-based approach will produce a result that converges to these definitions.

Throughout our definitions, we will refer to an angle as ω. In two dimensions, this refers to some angle θ. In three dimensions, this refers to a pair of angles, (θ, ϕ). Further, integrating a function, f, over ω, (i.e. $(\int f(\omega)d\omega)$ has different definitions for two and three dimensions. For two dimensions, this means $\int_0^{2\pi} f(\theta)d\theta$, while for three dimensions this means $\int_0^{2\pi}\int_0^{\pi} f(\theta,\phi) \times sin(\phi)d\phi d\theta$.

Utility Functions

We define the function $LENGTH(P, \omega)$ as the function that returns the length of the segment created by intersecting the object with a line going through point P at angle ω. As noted in the introduction, the intersection can create multiple, disconnected segments. We choose the length of the single segment that contains P for our definition. If P is not contained in the shape Q, then $LENGTH(P, \omega)$ returns 0.

We define the binary function $S_{le}(P, \omega, \alpha)$ as:

$$S_{le}(P,\omega,\alpha) = \begin{cases} 1 & P \epsilon Q \text{ and } LENGTH(P,\omega) \leq \alpha \\ 0 & \text{otherwise} \end{cases} \tag{8.1}$$

We also define a cumulative distribution function, $C_P(l)$ over a point P. Consider a random line L that intersects point P at angle ω. Then let $C_P(\alpha)$ be the probability that $LENGTH(P, \omega)$ is $\leq \alpha$. Then note that:

$$\frac{\int S_{le}(P,\omega,\alpha)d\omega}{\int 1 d\omega} = C_P(\alpha) \tag{8.2}$$

The left hand side of the equation is the proportion of angles that have $LENGTH(P, \omega)$ less than α, which is the same as $C_P(\alpha)$.

Volume Below α

Our first goal is to determine how much of the volume is at length scales $\leq \alpha$. This will be determined by integrating over all points P inside a shape Q. For each point, we must determine how many of the angles have chords that are at or below length α.

Then the formulation for $V_{le}(\alpha)$, the volume with length scale at or below α is:

$$V_{le}(\alpha) = \int_Q \frac{\int S_{le}(P,\omega,\alpha)d\omega}{\int 1 d\omega} dV \tag{8.3}$$

Substituting in equation 8.2 gives the final formula:

$$V_{le}(\alpha) = \int_Q C_P(\alpha) dV \tag{8.4}$$

Mass Below α

The mass from length scales at or below α, $M_{le}(\alpha)$, can be calculated by adding the density function, $\rho(P)$. Then:

$$M_{le}(\alpha) = \int_Q \rho(P) \frac{\int S_{le}(P, \omega, \alpha) d\omega}{\int 1 d\omega} dV \tag{8.5}$$

Again substituting using equation 8.2 gives:

$$M_{le}(\alpha) = \int_Q \rho(P) C_P(\alpha) dV \tag{8.6}$$

More Utility Functions

We now define additional utility functions that allow us to define mass and volume as a distribution over length scale.

Consider a random line L that intersects point P at angle ω. Then let $D_P(\alpha, \Delta)$ be the probability that $\alpha \leq LENGTH(P, \omega) \leq \alpha + \Delta$.

We define a new utility function, $S_{eq}(P, \omega, \alpha, \Delta)$, for calculating distributions of volume and mass:

$$S_{eq}(P, \omega, \alpha, \Delta) = \begin{cases} 1 & P \epsilon Q \text{ and } \alpha \leq LENGTH(P, \omega) \leq \alpha + \Delta \\ 0 & \text{otherwise} \end{cases} \tag{8.7}$$

Consider the integration of $S_{eq}(P, \omega, \alpha, \Delta)$ over all angles for some P and some α. Then:

$$\frac{\int S_{eq}(P, \omega, \alpha, \Delta) d\omega}{\int 1 d\omega} = D_P(\alpha, \Delta) \tag{8.8}$$

since the left-hand side of the equation is the proportion of angles that are at distances between α and $\alpha + \Delta$ from a boundary for point P, which is the same as $D_P(\alpha, \Delta)$.

Distribution Of Volume By Scale

Our goal is to construct a function $V_{eq}(\alpha, \Delta)$, where, for each α_0, $V_{eq}(\alpha_0, \Delta)$ is the amount of volume at length scales between α_0 and $\alpha_0 + \Delta$. One possible way to construct this function is to evaluate $V_{le}(\alpha_0 + \Delta) - V_{le}(\alpha_0)$. Alternatively, this can be constructed directly as:

$$V_{eq}(\alpha, \Delta) = \int_Q \frac{\int S_{eq}(P, \omega, \alpha, \Delta) d\omega}{\int 1 d\omega} dV \tag{8.9}$$

Substituting using equation 8.8 gives:

$$V_{eq}(\alpha, \Delta) = \int_Q D_P(\alpha, \Delta) dV \tag{8.10}$$

Distribution of Mass By Scale

Again, the mass can be calculated by adding a density function, $\rho(P)$:

$$M_{eq}(\alpha, \Delta) = \int_Q \rho(P) \frac{\int S_{eq}(P, \omega, \alpha, \Delta) d\omega}{\int 1 d\omega} dV \tag{8.11}$$

Again, substituting using equation 8.8 gives:

$$M_{eq}(\alpha, \Delta) = \int_Q \rho(P) D_P(\alpha, \Delta) dV \tag{8.12}$$

8.2 Implementation

All of the metrics can be calculated using a line scan approach. Each follows the same high level approach:

1. Construct a set of uniform density random lines
2. Calculate the intersection of these lines with the shape
3. Perform analysis on the resulting intersections

The only portion specific to each metric is step #3. Steps #1 and #2 are shared by all of the metrics.

Of course, parallelization aspects complicate this procedure. Here is an overview of the parallel approach:

1. Construct a set of uniform density, random lines. Each processor uses the exact same set of lines, regardless of which portion of the larger data set it operates on. (The technique for generating the lines will be discussed later in this chapter.)

2. Calculate the intersection of these lines with the shape. Each processor will only calculate the intersection with the portion of the data set it owns.

3. Re-distribute the segments. The random, uniform density lines will be partitioned among the processors. The intersections are communicated among the processors so that processor 0 contains all segments for the first N lines, processor 1 the segments for the next N lines, and so on. After redistribution has occurred, connected segments are identified and merged.

4. Perform analysis on the resulting intersections. Each processor will perform the analysis only on the lines it owns.

5. Collect results. All of the analyses are collected to processor 0, which merges them into a single result. The merge step is dependent on the analysis metric.

Figure 8.2 shows a diagram of this process. Finally, we should note that to get good convergence, millions of lines or more may be necessary. Because storing the intersections from this number of lines may exceed available memory, an iterative approach is taken where steps #1-#5 are performed repeatedly with tractable numbers of lines.

8.2.1 Constructing Uniform Density, Random Lines

Since we employ a probabilistic approach to evaluating these metrics, it is extremely important that the we do not choose lines that create bias. To do this, we choose a construction of lines that are random and uniform in density.

It is subtlely hard to generate a set of lines that meet this criteria. We used known techniques presented by Mazzolo and Roesslinger [47] for two dimensions and Sbert [65] for three dimensions. For axially-symmetric two-dimensional simulations, we used the three-

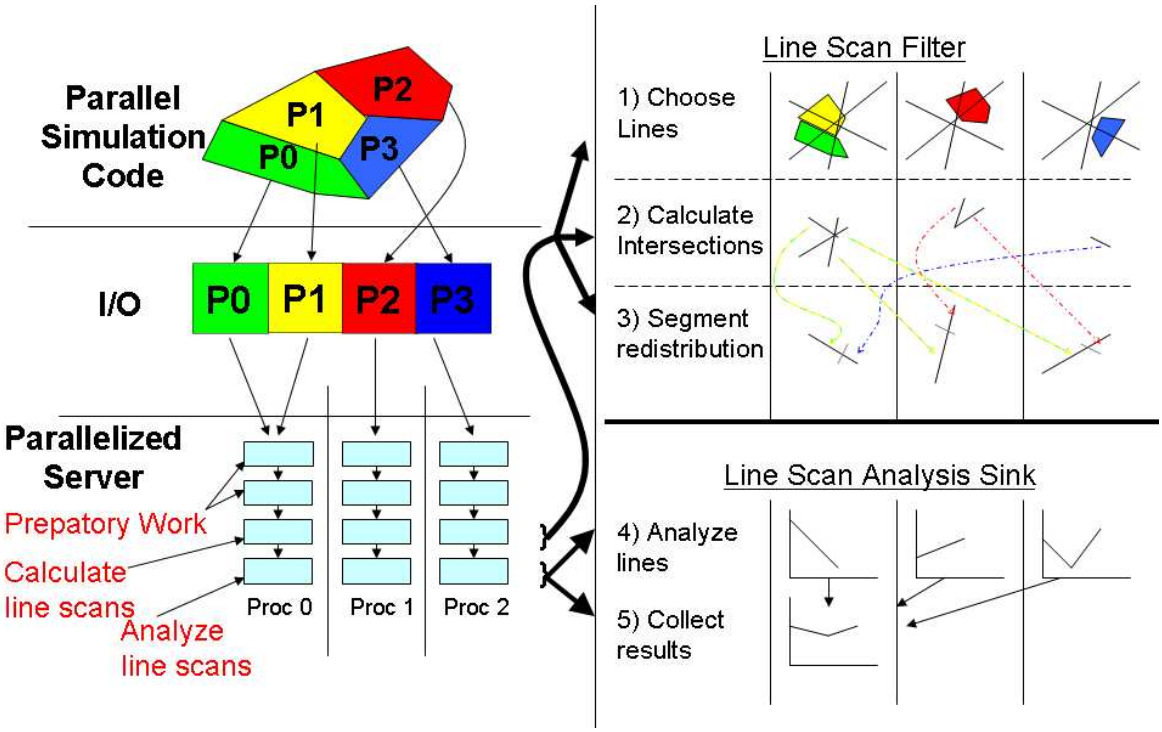

Figure 8.2: A diagram of data ownership for line scan-based analysis. Note that the data flow network has two filters at the beginning of the pipeline that are meant to be representative of filters that perform some data manipulation before the line scan-based analysis occurs. An example would be a filter that removes materials, leaving only one material in its place. This sort of interoperability between filters is a key advantage to deploying analysis techniques like line scan filters inside a greater, interoperable infrastructure.

dimensional lines and then considered their intersection with the axially-symmetric mesh when it is revolved into three dimensions (i.e. the Sbert construction was used).

The two dimensional construction circumscribes a circle around the objects. It starts by generating random, directed lines for a unit circle. We will need to consider *directed* lines so that, for a given point, we can generate the line at angle θ and also generate the line for angle $\theta + \pi$. Although these lines are essentially identical, we need to generate both directions so that we will be able to later correlate the lines to our definitions (which consider the angles θ and $\theta + \pi$ separately). The directed lines are then scaled by the radius of the circumscribing circle and translated with respect to its origin. To construct random directed lines for the unit circle, random values of p (distance to the origin) and θ (the angle between the line and the X-Axis) are selected. p is drawn uniformly from [-1, 1] and θ is drawn uniformly from [0, 2π). A segment is then constructed using p and θ.

One endpoint of the segment is the origin. The other endpoint, E, is distance p from the origin at angle θ. The random line is then the line perpendicular to this segment, passing through E. The line's equation is: $x \times cos(\theta) + y \times sin(\theta) = p$. The negative distances for p allow us to represent the line as originating from either direction. The direction is captured by retaining the θ parameter (the line is considered to be moving in the direction of $(sin(\theta), -cos(\theta))$[1]).

The three dimensional construction similarly circumscribes the object within a sphere. First, one point, P_0, on the surface of the sphere is chosen randomly. The point depends on the random selection of z from the interval $(-1, 1)$[2] and on the random selection of a θ from the interval $[0, 2\pi)$. Then P_0 is $(cos(\theta) \times \sqrt{1-z^2}, sin(\theta) \times \sqrt{1-z^2}, z)$. The normal vector, N, is the direction from the origin of the sphere to P_0, which is also $(cos(\theta) \times \sqrt{1-z^2}, sin(\theta) \times \sqrt{1-z^2}, z)$. A plane is constructed. The plane's origin is the origin of the sphere and its normal is N. The sphere is sliced by this plane to form a disc. From this disc, a random point is chosen. This is done by choosing two points, P_1 and P_2, and treating them as coordinates inside the disc. If (P_1, P_2) falls outside the disc, new P_1 and P_2 are chosen. The random directed line is then the line going in the direction of the normal vector and intersecting the disc at the point generated by (P_1, P_2). More detail of this process is located in Appendix A.

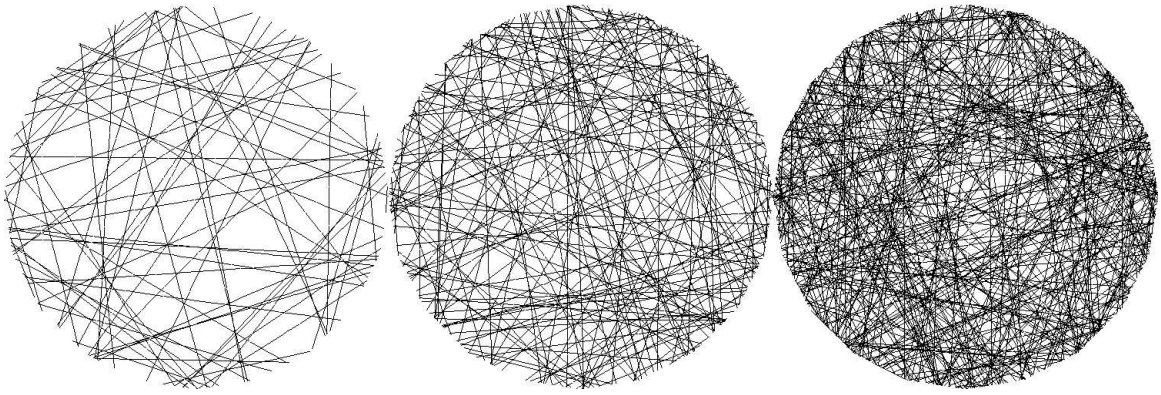

Figure 8.3: Random, uniform density lines within the unit circle. On the left, 125 lines, the middle, 250, and the right, 500.

[1]The direction of the line is not $(cos(\theta), sin(\theta))$, since the direction of the line is perpendicular to the angle θ. So the direction is $(cos(\theta - \frac{\pi}{2}), sin(\theta - \frac{\pi}{2}))$. $(cos(\theta - \frac{\pi}{2}) = sin(\theta)$ and $sin(\theta - \frac{\pi}{2}) = -cos(\theta).)$

[2]We choose from the open interval (as opposed to $[-1, 1]$), because of complexities discussed in Appendix A. Note that this choice precludes the points $(0, 0, -1)$ and $(0, 0, 1)$ from being selected.

8.2.2 Calculating Intersections

The difficult portion of line intersections is efficiently determining which elements of the mesh intersect with each of the lines. We do this by storing the bounding boxes of the elements in an interval tree [23]. For each line, the interval tree will return the elements whose bounding box intersects it. This approach can generate "false positive" elements, which have intersecting bounding boxes, but do not actually intersect the line. Empirically, their effect on running time is minimal.

A special case is for 2D axially-symmetric (i.e."RZ") meshes. These meshes should be treated as 3D meshes, through a revolution around the Z-Axis. However, truly promoting them into 3D by doing a revolution can create meshes bigger than can be held in primary memory. So our element identification routines and line intersections were modified so that the mesh could remain in 2D. In this case, the interval tree was modified to take a three dimensional line and operate on a two dimensional mesh. The elements of the mesh are grouped by spatial location. Each query to find the elements that intersect a line starts by taking the top level of the tree, revolving its bounding box into a hollow cylinder and asking if it can intersect the line. The tree is traversed using this test until the elements that can intersect are found. Once the correct elements are found, each of their edges are revolved to form truncated cones (or potentially cylindrical shells) and the intersection with the cone is determined. This intersection is then added to the line in the normal manner and the resulting grouping of line segments are in three dimensions (as opposed to projecting the three dimensional lines back to two dimensional curves).

We did consider taking two dimensional lines and weighting the lines so that the segment is adaptively weighted so that portions with high "R" values contribute more. But the resulting lines are no longer uniform density and random, so there is no reason to think that the resulting calculations will converge to the correct answer. An alternative approach would have been to cast curves on the plane. These curves would be the projection of the three dimensional line. But locating elements that intersect with curves and then calculating their intersections is also difficult and the approach taken was considered more straight forward.

8.2.3 Performing Analysis on the Resulting Intersections

Some of the considered metrics do their analysis on directed lines, while others do their analysis on rays (see table 8.2.3). For now, we will defer the construction of how to construct rays from directed lines, and instead focus on how the analysis is done given a set of lines or a set of rays. For the chord and ray length distributions, the probability distribution is discretized into bins to form a histogram. Every time a chord or ray is found, the bin for the corresponding length is incremented. Details of the length calculation depend on which variation of segment counting is used.

Table 8.1: A table of metrics and the primitive types (lines or rays) that they operate on.

Metric	Primitive
Chord length distribution (individual)	Lines
Chord length distribution (aggregate)	Lines
Ray length distribution (individual)	Rays
Ray length distribution (aggregate)	Rays
Volume as a function of length scale	Lines
Mass as a function of length scale	Lines

Now let us return to the topic of constructing random, uniform density rays from random, uniform density directed lines. There multiple ways to do this:

1. Find a random point along the line, going in the direction of the line.
2. Find multiple random points along the line, going in the direction of the line.
3. Consider all rays contained by the directed line.

Option #3 is the best option in this setting, because each line will contribute many data points to our probability distribution. Option #1, on the other hand, will possibly contribute no information for a given line, because the random point may not even lie inside the shape. Further, even if the randomly selected origin for the ray does lies in the shape, only a single data point will be inferred for our probability distribution.

Of course, care must be taken when considering all rays along a given directed line. There are an infinite number of rays. So we must consider what happens when we consider only rays spaced some ε apart and then consider what happens when ε approaches 0. Since

we have already decided to calculate our functions in a discretized form (i.e. a histogram), we can consider how many rays fall within a given bin. This will be the width of the bin divided by ε. However, since we are ultimately using these counts to form a distribution, the ε will cancel out and we can do our counting using lengths, rather numbers of rays (since there are an infinite number). For example, if our ray length distribution had 100 bins, each with width 3., and we encounter a line of length 57.3, we should update the first 19 bins with $\frac{3.}{\varepsilon}$. The $\frac{1.}{\varepsilon}$ term will ultimately cancel, so we can update the bin with simply a distance: 6. The 20^{th} bin should be updated with 0.3.

Although this construction is considerably more complicated, we will see in the verification section that it causes convergence to occur much more quickly.

The analysis done to calculate the mass as a function of length scale is quite involved and is discussed further in appendix B. Appendix B also contains the proof that a line scan-based technique will ultimately converge to the same solution.

8.3 Verification

All of the metrics we implemented leveraged the same line scan infrastructure. Since the metrics are not aware of the underlying shapes, applying the metrics to well known shapes, such as spheres and circles, is a sufficient way to test that infrastructure. In addition, analytic answers for these shapes are well known, which allows us to verify the answers the metrics produce. In this section we will test the metrics against a sphere.

8.3.1 Test Data

The sphere itself was created by taking a rectilinear grid and clipping it to create a sphere of radius 1. The zones on the interior of the grid were then "voxels" (hexahedrons with axis-aligned faces), while the zones on the sphere's boundary were tetrahedra created by subdividing these voxels. The total number of zones was 36,000. Although some of the metrics do weighting based on density, a uniform density was used to ease comparisons with analytic results.

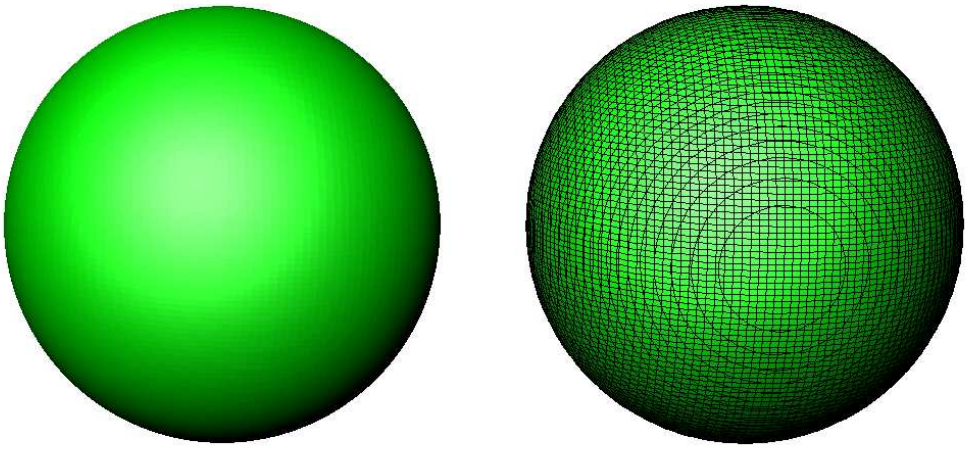

Figure 8.4: The sphere is rendered by itself on the left and mesh lines are added on the right. There are 36,000 total zones in the sphere.

8.3.2 Chord Length Distribution

Olson, Miller, et al, calculated the analytic solution for the chord length distribution of a sphere of radius R in [57]. The distribution is:

$$\theta_C(l) = \begin{cases} 0 & l > 2R \\ \frac{l}{2R^2} & l \leq 2R \end{cases} \tag{8.13}$$

Of course, our approach to calculate the chord length distribution was to cast some number, L, of uniform density, random lines and to calculate the probability distribution over those lines, $\theta_{C,L}(l)$. We then studied the resulting distributions with 10,000, 100,000, and 1,000,000 lines. For each amount of lines, we used the following metrics to measure closeness of fit for $\theta_{C,L}(l)$ with $\theta_C(l)$:

- The L2-norm, $\int_0^{2R} (\theta_{C,L}(l) - \theta_C(l))^2 dl$
- The difference in average chord length, $\int_0^{2R} l * \theta_{C,L}(l) dl - \int_0^{2R} l * \theta_C(l) dl$

For each run, we used 250 bins. Table 8.2 and figures 8.5, 8.6, and 8.7 contain the results. As more lines were cast, the two metrics above give better answers. It is also worth noting that convergence is dependent on bin-size. Obviously, wide bin sizes converge much more quickly that narrow bin sizes. Further, when using a fixed bin size, the result cannot ever exactly equal the analytic solution, because the histogram produced only approximates the curve given by the analytic solution.

Table 8.2: Summarizing convergence for different numbers of lines for the chord length distribution.

	Analytic	10K	100K	1M
L2-norm	0	0.1715	0.0472	0.0326
Avg chord length	1.3323	1.3188	1.3305	1.3318
Diff in avg chord length	0	0.0135	0.0018	0.0005
Time to calculate	NA	18 sec	3 minutes	25 minutes

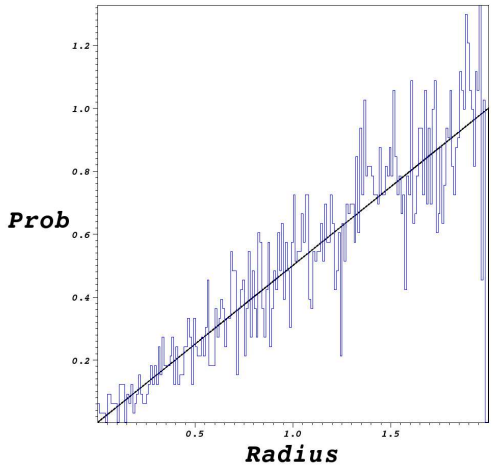

Figure 8.5: Plotting the distribution produced by casting 10,000 lines (blue) versus the analytic solution (black).

8.3.3 Ray Length Distribution

For the ray length distribution, we again did a comparison between the analytic and calculated results. This time we only used 1,000, 10,000, and 100,000 lines, since convergence was considerably better. We used the same metrics to compare the results:

- $\int_0^{2R} l * \theta_{R,L}(l)dl - \int_0^{2R} l * \theta_R(l)dl$
- $\int_0^{2R} (\theta_{R,L}(l) - \theta_R(l))^2 dl$

Again, for each run, we used 250 bins and saw good convergence to the analytic solution. Table 8.3 and figures 8.8, 8.9, and 8.10 contain the results.

8.3.4 Demonstration

We will now study the information given by our metrics for a grain-like media, as shown in figure 8.11A. In the rest of figure 8.11, we show the outputs of each metric on

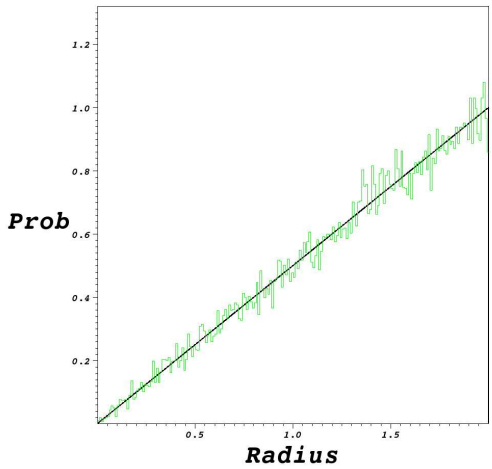

Figure 8.6: Plotting the distribution produced by casting 100,000 lines (green) versus the analytic solution (black).

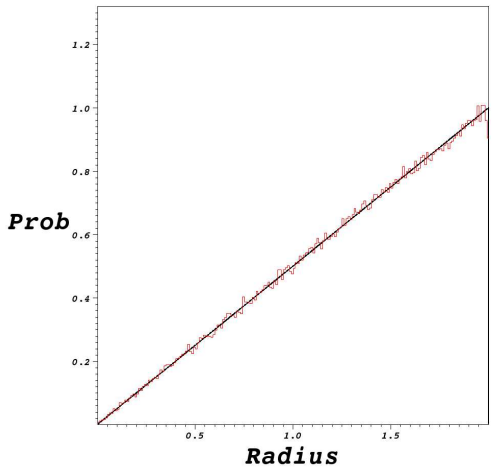

Figure 8.7: Plotting the distribution produced by casting 1,000,000 lines (red) versus the analytic solution (black).

these shapes. The aggregate chord length distribution (B) gives a probability (over lengths) of how much media is traversed when starting from outside the shape and moving through the entire shape. The individual chord length distribution (C) gives the probability of the length of a randomly selected chord. The mass distribution (D) shows how much mass is at each length scale. Since the media had uniform density, the mass distribution is equal to the volume distribution in this case. The aggregate ray length distribution (E) gives the probability of how much media a particle traverses to exit the entire shape, provided that the particle was randomly placed and going a random direction. The individual ray length distribution (F) gives the probability of how much media a particle traverses to exit the

Table 8.3: Summarizing convergence for different numbers of lines for the ray length distribution.

	Analytic	1K	10K	100K
L2-norm	0	0.0120	0.0033	0.0017
Avg chord length	0.75	0.7497	0.75009	0.7490
Diff in avg chord length	0	0.00133	0.00009	0.0010
Time to calculate	NA	3 sec	22 sec	3 minutes

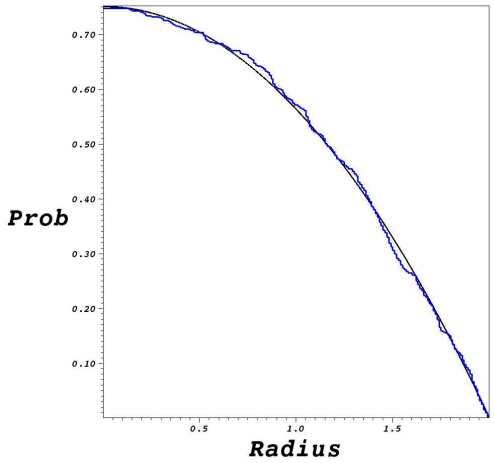

Figure 8.8: Plotting the ray length distribution produced by casting 1,000 lines (blue) versus the analytic solution (black).

portion of the shape that it initially lies in, again provided that the particle was randomly placed and going a random direction.

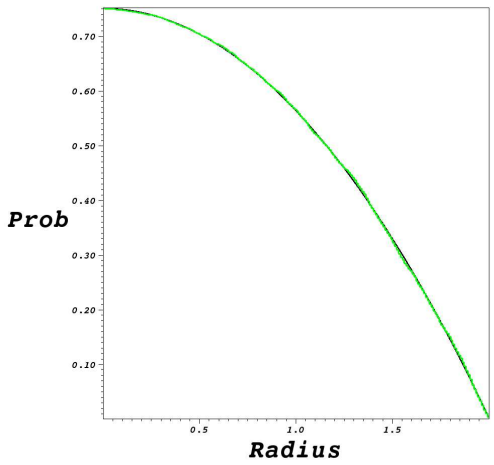

Figure 8.9: Plotting the ray length distribution produced by casting 10,000 lines (green) versus the analytic solution (black).

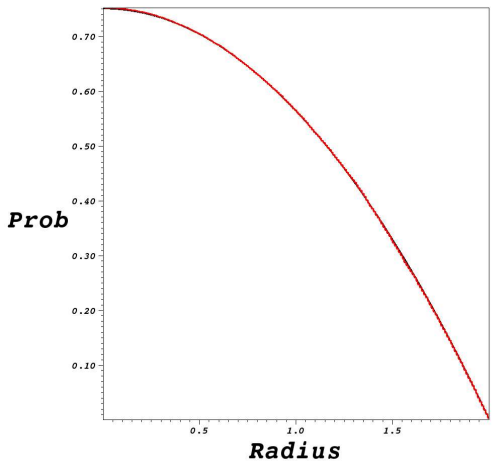

Figure 8.10: Plotting the ray length distribution produced by casting 100,000 lines (red) versus the analytic solution (black).

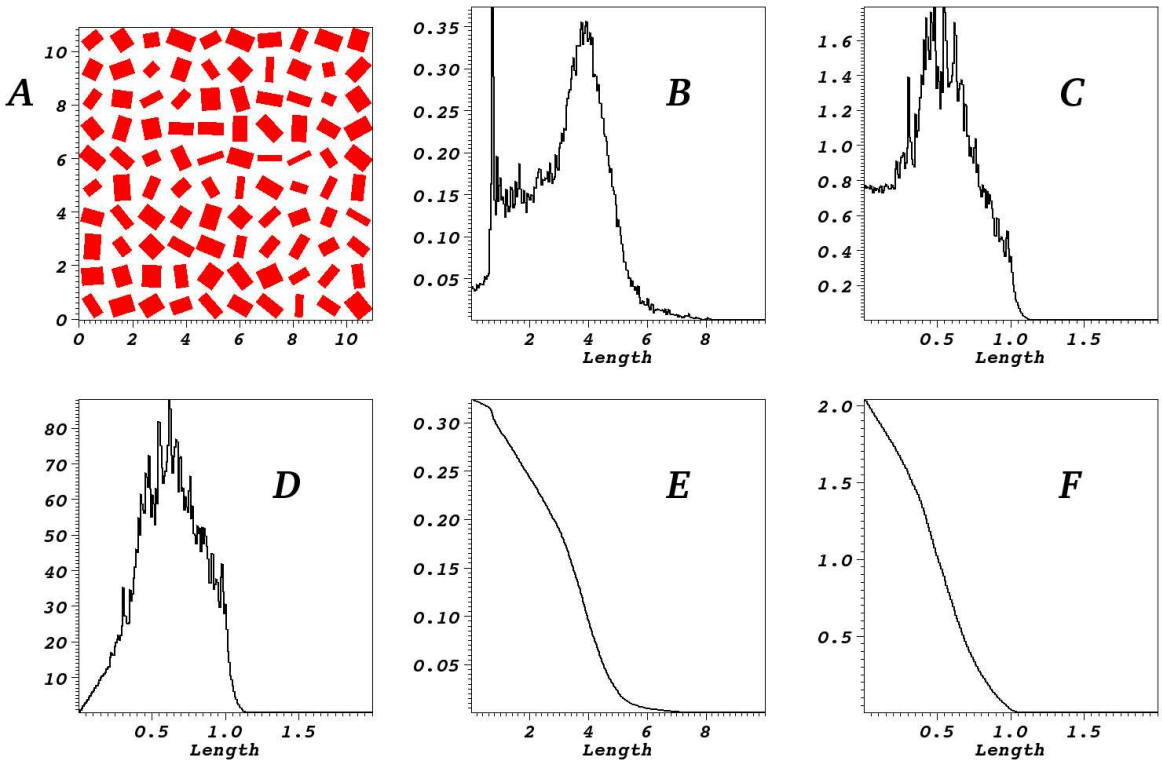

Figure 8.11: The shape characterization metrics were applied to the shape shown in A. The aggregate chord length distribution is plotted in B. The individual chord length distribution is plotted in C. The mass distribution is plotted in D. The aggregate ray length distribution is plotted in E. The individual ray length distribution is plotted in F.

Bibliography

[1] Greg Abram and Lloyd A. Treinish. An extended data-flow architecture for data analysis and visualization. Research report RC 20001 (88338), IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, February 1995.

[2] A. Agresti. *Categorical Data Analysis. Second Edition.* Wiley-Interscience, 2002.

[3] James Ahrens, Kristi Brislawn, Ken Martin, Berk Geveci, C. Charles Law, and Michael Papka. Large-scale data visualization using parallel data streaming. *IEEE Comput. Graph. Appl.*, 21(4):34–41, 2001.

[4] Gabrielle Allen, Werner Benger, Thomas Dramlitsch, Tom Goodale, Hans-Christian Hege, Gerd Lanfermann, Andre Merzky, Thomas Radke, Edward Seidel, and John Shalf. Cactus tools for grid applications. *Cluster Computing*, 4(3):179–188, 2001.

[5] Rajesh K. Batra and Lambertus Hesselink. Feature comparisons of 3-D vector fields using earth mover's distance. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 105–114, 1999.

[6] J. Baum. *Elements of Point Set Topology.* Dover Publications, 1991.

[7] L. Bavoil, S. Callahan, P. Crossno, J. Freire, C. Scheidegger, C. Silva, and H. Vo. Vistrails: Enabling interactive multiple-view visualizations. In *VIS '05: Proceedings of the conference on Visualization '05*, pages 135–142, 2005.

[8] R. Becker and W. Cleveland. Brushing scatterplots. *Technometrics*, 29(2):127–142, 1987.

[9] Wes Bethel, Brian Tierney, Jason Lee, Dan Gunter, and Stephen Lau. Using high-speed wans and network data caches to enable remote and distributed visualization. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 28, Washington, DC, USA, 2000. IEEE Computer Society.

[10] Kathleen S. Bonnell, Mark A. Duchaineau, Daniel R. Schikore, Bernd Hamann, and Kenneth I. Joy. Material interface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):500–511, 2003.

[11] Kathleen S. Bonnell, Kenneth I. Joy, Bernd Hamann, Daniel R. Schikore, and Mark Duchaineau. Constructing material interfaces from data sets with volume-fraction information. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 367–372, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[12] D. Butler and M Pendley. A visualization model based on the mathematics of fiber bundles. *Computers in Physics*, 3(5):45–51, 1989.

[13] William H. Cabot and Andrew W. Cook. Reynolds number effects on rayleigh-taylor instability with possible implications for type-ia supernovae. *Nature: Physics*, 2(8):562–568, August 2006.

[14] Michael Cannon and Tony Warnock. A shape decriptor based on the line scan transform. Technical Report LA-UR-04-5865, Los Alamos National Laboratory, 2004.

[15] Xavier Cavin, Christopher Mion, and Alain Filbois. Cots cluster-based sort-last rendering: Performance evaluation and piplined implementation. In *VIS '05: Proceedings of the conference on Visualization '05*, pages 111–118, 2005.

[16] Yi-Jen Chiang and Claudio T. Silva. I/o optimal isosurface extraction (extended abstract). In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 293–ff. IEEE Computer Society Press, 1997.

[17] Hank Childs, Eric Brugger, Kathleen Bonnell, Jeremy Meredith, Mark Miller, Brad Whitlock, and Nelson Max. A contract based system for large data visualization. In *VIS '05: Proceedings of the conference on Visualization '05*, 2005.

[18] Hank Childs, Mark A. Duchaineau, and Kwan-Liu Ma. A scalable, hybrid scheme for volume rendering massive data sets. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 153–162, 2006.

[19] Hank Childs and Mark Miller. Beyond meat grinders: An analysis framework addressing the scale and complexity of large data sets. In *SpringSim High Performance Computing Symposium (HPC 2006)*, pages 181–186, 2006.

[20] J. Clyne and M. Rast. A prototype discovery environment for analyzing and visualizing terascale turbulent fluid flow simulations. In *In Proceedings of SPIE Visualization and Data Analysis 2005*, 2005.

[21] Computational Engineering International, Inc. *EnSight User Manual*, May 2003.

[22] Rich Cook, Nelson Max, Claudio Silva, and Peter Williams. Image-space visibility ordering for cell project volume rendering of unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 10(6):695–707, November 2004.

[23] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, 1990.

[24] Willem C. de Leeuw, Hans-Georg Pagendarm, Frits H. Post, and Birgit Walter. Visual simulation of experimental oil-flow visualization by spot noise images from numerical flow simulation. In *Visualization in Scientific Computing '95*, pages 135–148, 1995.

[25] D. Demarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen. Distributed interactive ray tracing for large volume visualization. In *In Proceedings of the Symposium on Parallel and Large-Data Visualization and Graphics*, pages 87–94, 2003.

[26] Herbert Edelsbrunner, John Harer, Vijay Natarajan, and Valerio Pascucci. Local and global comparison of continuous functions. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 275–280, October 2004.

[27] L. Freitag and T. Urness. Analyzing industrial furnace efficiency using comparative visualization in a virtual reality environment. Technical Report ANL/MCS-P744-0299, Argonne National Laboratory, 1999.

[28] Jinzhu Gao, Jian Huang, C. Ryan Johnson, and Scott Atchley. Distributed data management for large volume visualization. In *VIS '05: Proceedings of the conference on Visualization '05*, pages 183–190, 2005.

[29] G.A. Geist, J.A. Kohl, and P.M. Papadopoulos. Cumulvs: Providing fault-tolerance, visualization and steering of parallel applications. *SIAM*, August 1996.

[30] D. Gresh, B. Rogowitz, R. Winslow, D. Scollan, and C. Yung. Weave: A system for visually linking 3-d and statistical visualizations, applied to cardiac simulation and measurement data. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 489–492, 2000.

[31] R. Haimes. pv3: A distributed system for large-scale unsteady CFD visualization. In *In AIAA 32nd Aerospace Scienced Meeting and Exhibit, number AIAA 94-0321*, 1994.

[32] R. Haimes and D. Edwards. Visualization in a parallel processing environment. In *American Institute of Aeronautics and Astronautics*, 1997.

[33] Matthias Hopf and Thomas Ertl. Hierarchical splatting of scattered data. In *VIS '03: Proceedings of the conference on Visualization '03*, pages 443–440, 2003.

[34] Jinhee Jeong and Fazle Hussain. On the identification of a vortex. *Journal of Fluid Mechanics*, pages 69–94, 285 1995.

[35] C.R. Johnson, S. Parker, and D. Weinstein. Large-scale computational science applications using the SCIRun problem solving environment. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[36] R. Kirby, H. Marrmanis, and D. Laidlaw. Visualizing multivalued data from 2d incompressible flows using concepts from painting. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 333–340, 1999.

[37] Kitware, Inc. *The Visualization Toolkit User's Guide*, January 2003.

[38] Edwin J. Kokko, Harry E. Martz, Diane J. Chinn, Hank R. Childs, Jessie A. Jackson, David H. Chambers, Daniel J. Schneberk, and Grace A. Clark. As-built modeling of objects for performance assessment. *Journal of Computing and Information Science in Engineering*, 6(4):405–417, 12 2006.

[39] C. Charles Law, Amy Henderson, and James Ahrens. An application architecture for large data visualization: a case study. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 125–128. IEEE Press, 2001.

[40] Steve M. Legensky. Interactive investigation of fluid mechanics data sets. In *VIS '90: Proceedings of the 1st conference on Visualization '90*, pages 435–439. IEEE Computer Society Press, 1990.

[41] Eric Lum and Kwan-Liu Ma. Hardware-accelerated parallel non-photorealistic volume rendering. In *Proc. NPAR 2002*, pages 67–74, 2002.

[42] Eric Lum, Brett Wilson, and Kwan-Liu Ma. High-quality lighting and efficient pre-integration for volume rendering. In *Proc. of Joint Eurographics-IEEE TVCG Symposium on Visualization*, pages 25–34, 2004.

[43] Kwan-Liu Ma and David M. Camp. High performance visualization of time-varying volume data over a wide-area network status. In *Proceedings of Supercomputing*, 2000.

[44] Kwan-Liu Ma and Tom Crockett. A scalable parallel cell-projection volume rendering algorithm for 3d unstructured data. In *Proc. 1997 Symposium on Parallel Rendering*, pages 95–104, 1997.

[45] Kwan-Liu Ma, Eric B. Lum, Hongfeng Yu, Hiroshi Akiba, Min-Yu Huang, Min-Yu Huang, Yue Wang, and Greg Schussm. Scientific discovery through advanced visualization. DOE SciDAC 2005 Conference, June 2005.

[46] Nelson Max, Peter Williams, Claudio Silva, and Richard Cook. Volume rendering for curvilinear and unstructured grids. In *Computer Graphics International 2003*, pages 210–215, 2003.

[47] Alain Mazzolo and Benoit Roesslinger. Monte-carlo simulation of the chord length distribution function across convex bodies, non-convex bodies and random media. *Monte Carlo Methods and Applications*, 10:443–454, 2004.

[48] Alain Mazzolo, Benoit Roesslinger, and Cheikh Diop. On the properties of the chord length distribution, from integral geometry to reactor physics. *Annals of Nuclear Energy*, 30(14):1391–1400, 2003.

[49] Alain Mazzolo, Benoit Roesslinger, and Wilfried Gille. Properties of chord length distributions of nonconvex bodies. *Journal of Mathematical Physics*, 44(12):6195–6208, 2003.

[50] Patrick S. McCormick, Jeff Inman, James P. Ahrens, Charles Hansen, and Greg Roth. Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 171–178, Washington, DC, USA, 2004. IEEE Computer Society.

[51] M. Miller, C. Hansen, and C.R. Johnson. Simulation steering with SCIRun in a distributed environment. In B. Kstrm, J. Dongarra, E. Elmroth, and J. Wasniewski, editors, *Applied Parallel Computing, 4th International Workshop, PARA'98*, volume 1541 of *Lecture Notes in Computer Science*, pages 366–376. Springer-Verlag, Berlin, 1998.

[52] Mark C. Miller, James F. Reus, Robb P. Matzke, William J. Arrighi, Larry A. Schoof, Ray T. Hitt, and Peter K. Espen. Enabling interoperation of high performance, scientific computing applications: Modeling scientific data with the sets & fields (saf) modeling system. In *ICCS '01: Proceedings of the International Conference on Computational Science-Part II*, pages 158–170, London, UK, 2001. Springer-Verlag.

[53] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, 1994.

[54] Patrick J. Moran and Chris Henze. Large field visualization with demand-driven calculation. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 27–33, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[55] Kenneth Moreland, Brian Wylie, and Constantine Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 85–92, Piscataway, NJ, USA, 2001. IEEE Press.

[56] Network Theory Ltd. *The Python Language Reference Manual*, 2003.

[57] Gordon L. Olson, David S. Miller, Edward W. Larsen, and Jim E. Morel. Chord length distributions in binary stochastic media in two and three dimensions. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 101(2):269–283, 2006.

[58] Valerio Pascucci and Randall J. Frank. Global static indexing for real-time exploration of very large regular grids. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 2–2, New York, NY, USA, 2001. ACM Press.

[59] Valerio Pascucci, Daniel E. Laney, Randall J. Frank, Giorgio Scorzelli, Lars Linsen, Bernd Hamann, and Francois Gygi. Real-time monitoring of large scientific simulations. In H. Haddad and G. Papadopoulos, editors, *Proceedings of ACM Symposium on Applied Computing (SAC 2003)*, page 10. ACM Press, 2003.

[60] Brian Paul. *The Mesa 3-D graphics library*, 1996. http://www.mesa3d.org.

[61] K Perlen and E.M. Hoffert. Hypertexture. *Computer Graphics*, 23(3):253–261, July 1989.

[62] Thomas Porter and Tom Duff. Compositing digital images. *Computer Graphics*, 18(3):253–259, 1984.

[63] Lisa Roberts. *Silo User's Guide*. Lawrence Livermore National Laboratory, 2000. UCRL-MA-118751-REV-1.

[64] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 97–108, August 2000.

[65] Mateu Sbert. *The Use of Global Random Directions to Compute Radiosity. Global Monte Carlo Techniques*. PhD dissertation, Universitat Politecnica de Catalunya, Department de LLenguatges i Sistemes Informatics, November 1996.

[66] Larry Schoof and Vincent Yarberry. *EXODUS II: A Finite Element Data Model*. Sandia National Laboratory, 1994. Tech Rep. SAND92-2137.

[67] William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *VIS '96: Proceedings of the conference on Visualization '96*, pages 93–ff. IEEE Computer Society Press, 1996.

[68] D.W. Scott. *Multivariate Density Estimation: Theory, Practice and Visualization*. Wiley & Sons, 1992.

[69] Qin Shen, Alex Pang, and Sam Uselton. Data level comparison of wind tunnel and computational fluid dynamics data. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 415–418, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[70] Qin Shen, Sam Uselton, and Alex Pang. Comparison of wind tunnel experiments and computational fluid dynamics simulations. *Journal of Visualization*, 6(1):31–39, 2003.

[71] C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. In *Visualization 2002 Course Notes*, 2002.

[72] O. Sommer and T. Ertl. Comparative visualization of instabilities in crash-worthiness simulations. In *Procceedings of EG/IEEE TCVG Symposium on Visualization VisSym '01*, pages 319–328, 2001.

[73] Sherman K. Stein and Anthony Barcellos. *Calculus and Analytic Geometry*. McGraw-Hill, Inc, 1992.

[74] K. Stockinger, J. Shalf, K. Wu, and W. Bethel. Query-driven visualization of large data sets. In *VIS '05: Proceedings of the conference on Visualization '05*, pages 167–174, 2005.

[75] Magnus Strengert, Marcelo Magallon, Daniel Weiskopf, Stefan Guthe, and Thomas Ertl. Hierarchical visualization and compression of large volume datasets using gpu clusters. In *Proc. Eurographics/ACM SIGGRAPH Parallel Graphics and Visualization 2004*, pages 41–48, 2004.

[76] *The Top 500 Supercomputers*, 2006. http://www.top500.org.

[77] Jens Trapp and Hans-Georg Pagendarm. Data level comparative visualization in aircraft design. In *VIS '96: Proceedings of the conference on Visualization '96*, pages 393–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[78] Craig Upson, Thomas Faulhaber Jr., David Kamins, David H. Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, and Andries van Dam. The application visualization system: A computational environment for scientific visualization. *Computer Graphics and Applications*, 9(4):30–42, July 1989.

[79] Vivek Verma and Alex Pang. Comparative flow visualization. *IEEE Trans. Vis. Comput. Graph.*, 10(6):609–624, 2004.

[80] Chaoli Wang, Jinzhu Gao, and Han-Wei Shen. Parallel multiresolution volume rendering of large data sets with error-guided load balancing. In *Proc. Eurographics/ACM SIGGRAPH Parallel Graphics and Visualization 2004*, pages 23–30, 2004.

[81] M. Ward. Xmdvtool: Integrating multiple methods for visualizing multivariate data. In *VIS '94: Proceedings of the conference on Visualization '94*, pages 326–336, 1994.

[82] Gunther H. Weber, Oliver Kreylos, Terry J. Ligocki, John M. Shalf, Hans Hagen, Bernd Hamann, and Kenneth I. Joy. Extraction of crack-free isosurfaces from adaptive mesh refinement data. In *Approximation and Geometrical Methods for Scientific Visualization*, editors Gerald Farin, Hans Hagen, and Bernd Hamann, pages 19-40, 2003.

[83] E.J. Wegman. Data compression by quantization. http://www.galaxy.gmu.edu/interface/I01/I2001Proceedings/EWegman/Data-CompressionbyQuantization.pdf.

[84] Lance Williams. Casting curved shadows on curved surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274, New York, NY, USA, 1978. ACM Press.

[85] Peter L. Williams and Samual P. Uselton. Foundations for measuring volume rendering quality. Technical Report NAS/96-021, NASA Numerical Aerospace Simulation, 1996.

[86] R. Yagel, D. M. Reed, A. Law, P. Shih, and N. Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing. In *Proceedings 1996 Symposium on Volume Visualization*, pages 55–62, 1996.

[87] Hongfeng Yu, Kwan-Liu Ma, and Joel Welling. I/o strategies for parallel rendering of large time-varying volume data. In *Proc. Eurographics/ACM SIGGRAPH Parallel Graphics and Visualization 2004*, pages 31–40, 2004.

[88] Hualin Zhou, Min Chen, and Mike F. Webster. Comparative evaluation of visualization and experimental results using image comparison metrics. In *VIS '02: Proceedings of the conference on Visualization '02*, Washington, DC, USA, 2002. IEEE Computer Society.

[89] http://www.galaxy.gmu.edu/stats/syllabi/inft979/GeometricQuantization.pdf.

Appendix A

A Mapping Function Between Line Scans and Volumes

In this chapter, we will establish a mapping function between the random, uniform density directed lines from chapter 8 and the definitions also from that chapter. Specifically, we will prove properties of this mapping function that will be used in Appendix B for translating analysis on these lines to the definitions (which are in a point-angle space).

For ease of reference, let's name the two sets that we are considering. The first set, *Lines*, is over all points on all directed line segments contained inside a unit circle or unit sphere. Elements of this set are a tuple. An example from this set would be (m, t), where m is a directed line and t is a distance along that line. The second set, *PointAngle*, was previously represented in the definitions from Chapter 8 as "$d\omega dV$". An element from *PointAngle* in two dimensions is of the form (x, y, θ). In three dimensions, it is of the form (x, y, z, θ, ϕ). For a tuple to be in *PointAngle*, its spatial coordinates must be within the unit circle or unit sphere and its angle (or angles) must be constrained so that no point and angle pairing can be multiply represented (i.e. θ must be from the interval $[0, 2\pi)$ and ϕ must be from the interval $[0, \pi]$). All of the derivations in this chapter assume the sets *Lines* and *PointAngle* are over the unit circle or unit sphere. If a shape is considered that is outside this region, the region can be scaled and shifted appropriately.

Again, our ultimate goal is to show that analyses done on the set *Lines* are ap-

plicable to the set *PointAngle*. The bulk of this chapter will be establishing a mapping function, F, that will help establish this. We will consider the cases of two dimensional directed lines and three dimensional directed lines separately. For each dimension, we will derive:

- the definition of F
- whether or not F is a one-to-one correspondence, and
- the Jacobian of F.

A.1 Relating results in *Lines* to *PointAngle*

In [73], a theorem is established that links evaluations from one set with evaluations on another. If F is a one-to-one mapping function from set B onto set A, and if h is a function defined on A, then:

$$\int_A h(a)dA = \int_B h(F(b)) \times J_F(b)dB \tag{A.1}$$

where $J_F(b)$ is the determinant of the Jacobian of F at point b.

In appendix B, we will use this theorem to construct a relationship between our sets *Lines* and *PointAngle*. *PointAngle* will be set A and *Lines* will be set B. F will be the mapping function established in the following sections.

For three dimensions, the mapping function F will not form a one-to-one correspondence. Part of the three dimensional construction of random directed lines involves choosing a point on the surface of the sphere. The construction chooses a random z and a random θ. If z is -1 or 1, then, for this construction, any choice of θ will yield the same point (making the function not one-to-one). For this reason, we only draw z from the open interval $(-1, 1)$. Hence, the integration on the right hand side of A.1 will exclude the boundary of the set *Lines*. Similarly, the image of F will be over an open subset of *PointAngle*, where the angles ϕ equal to 0 and π are not represented. Later, in appendix B we will show that the integrand in the excluded region is finite. Then, since the boundary of *Lines* and its image in *PointAngle* are submanifolds of lower dimensions (and the

integrand is finite), the excluded regions do not contribute to the integral and theorem A.1 will still be applicable.

A.2 Two dimensional directed lines

The two dimensional directed line segment construction calls for choosing a p randomly from the interval $[-1, 1]$ and a θ_L randomly from the interval $[0, 2 \times \pi)$[1]. For a given (p, θ_L) pair, the first step is to construct the line:

$$x \times cos(\theta_L) + y \times sin(\theta_L) = p \tag{A.2}$$

and then retain θ_L to prescribe the direction. Intersecting this directed line with the unit circle creates a direted line segment. It is this segment that is a member of $Lines_{2D}$.

A.2.1 The mapping function F

Consider the function F that maps elements from $Lines_{2D}$ to $PointAngle_{2D}$. Previously, we denoted a directed line as m. Since the construction of the directed line m is dependent on the parameters p and θ_L, we will include both parameters, p and θ_L, in the place of m. That is, instead of considering $F(m, t) \rightarrow (x, y, \theta_{PA})$, we will consider $F(p, \theta_L, t) \rightarrow (x, y, \theta_{PA})$.

We will now derive x, y, and θ_{PA} from p, θ_L, and t. First, we will determine the endpoints of the line segment formed by the (p, θ_L) construction from A.2. Re-arranging the equation to solve for x gives:

$$x = \frac{p - y \times sin(\theta_L)}{cos(\theta_L)} \tag{A.3}$$

We also know that the endpoints will lie on the unit circle, so:

$$x^2 + y^2 = 1 \tag{A.4}$$

Substituting A.3 into A.4 gives:

$$\frac{p^2 - 2py \times sin(\theta_L) + y^2 \times sin(\theta_L)}{cos^2(\theta_L)} + y^2 = 1 \tag{A.5}$$

[1]The subscript "L" for θ_L is a mnemonic for the set "*Lines*". Subsequently, we will also discuss the related angle θ_{PA}, which is the θ parameter for set "*PointAngle*".

Rearranging and using the identity $sin^2(\theta_L) + cos^2(\theta_L) = 1$ gives:

$$y^2 - 2p \times sin(\theta_L) + (p^2 - cos^2(\theta_L)) = 0 \tag{A.6}$$

And we can use the quadratic formula to obtain the roots for y:

$$y_1 = p \times sin(\theta_L) + cos(\theta_L) \times \sqrt{1 - p^2} \tag{A.7}$$

$$y_2 = p \times sin(\theta_L) - cos(\theta_L) \times \sqrt{1 - p^2} \tag{A.8}$$

Finally, we can solve for x_1 and x_2 by substituting A.7 and A.8 into A.3, to get:

$$x_1 = p \times cos(\theta_L) - sin(\theta_L) \times \sqrt{1 - p^2} \tag{A.9}$$

$$x_2 = p \times cos(\theta_L) + sin(\theta_L) \times \sqrt{1 - p^2} \tag{A.10}$$

Note that (x_1, y_1) can be thought of as the first point from the unit circle that the directed line will hit and (x_2, y_2) can be thought of the last point from the unit circle it will hit.

We will now construct the mapping $F(P, \theta_L, t) \rightarrow (x, y, \theta_{PA})$. We would like to determine the form of the following:

$$x = F_x(P, \theta_L, t) \tag{A.11}$$

$$y = F_y(P, \theta_L, t) \tag{A.12}$$

$$\theta_{PA} = F_{\theta_{PA}}(P, \theta_L, t) \tag{A.13}$$

Using the construction for a random p and θ_L, the directed line segment originates at (x_1, y_1) and terminates at (x_2, y_2). Consider a point on this segment. It starts at (x_1, y_1) and goes some distance along the vector $(x_2 - x_1, y_2 - y_1)$. Call the unit vector along this orientation (x_d, y_d). Then we know that:

$$x = x_1 + t \times x_d \tag{A.14}$$

$$y = y_1 + t \times y_d \tag{A.15}$$

The unit vector, (x_d, y_d), can be calculated by dividing $(x_2 - x_1, y_2 - y_1)$ by its magnitude. This magnitude is calculated as follows. A right triangle is formed by the perpendicular line and the segment of length p that originates from the origin (see figure A.1). The hypotenuse

is 1, so the third side, using the Pythagorean Theorem, is $\sqrt{1-p^2}$. This third side accounts for half the length of the chord $(x_2 - x_1, y_2 - y_1)$, so its total length is $2\sqrt{1-p^2}$. Then (x_d, y_d) is:

$$x_d = \frac{x_2 - x_1}{2\sqrt{1-p^2}} = \frac{2 \times sin(\theta_L) \times \sqrt{1-p^2}}{2\sqrt{1-p^2}} = sin(\theta_L) \tag{A.16}$$

$$y_d = \frac{y_2 - y_1}{2\sqrt{1-p^2}} = \frac{-2 \times cos(\theta_L) \times \sqrt{1-p^2}}{2\sqrt{1-p^2}} = -cos(\theta_L) \tag{A.17}$$

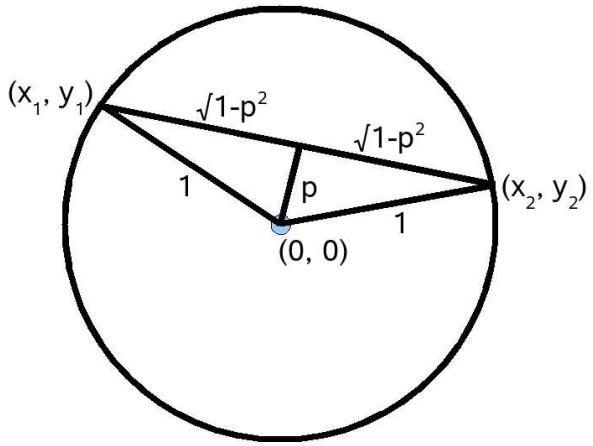

Figure A.1: Establishing the length of the chord.

We can now get an explicit formula for x and y by substituting A.16 and A.17 into A.14 and A.15:

$$x = p \times cos(\theta_L) - sin(\theta_L) \times \sqrt{1-p^2} + t \times sin(\theta_L) \tag{A.18}$$

$$y = p \times sin(\theta_L) - cos(\theta_L) \times \sqrt{1-p^2} - t \times cos(\theta_L) \tag{A.19}$$

Consider the relationship between θ_L and θ_{PA}. The directed line formed in the construction is perpendicular to θ_L, so it is at a rotation of $\frac{-\pi}{2}$. So:

$$\theta_{PA} = \begin{cases} \theta_L - \frac{\pi}{2} & \theta_L > \frac{\pi}{2} \\ \theta_L + \frac{3\pi}{2} & \text{otherwise} \end{cases} \tag{A.20}$$

Although we have given the equations for the mapping F, we have not yet shown that it is a mapping from $Lines_{2D}$ to $PointAngle_{2D}$, because we have not shown that all

elements from $Lines_{2D}$ only map to elements in $PointAngle_{2D}$. (Restated, there might exist an element of $Lines_{2D}$ that is mapped to a tuple that is not an element of $PointAngle_{2D}$.) Assume, then, that there exists an element, E, from $Lines_{2D}$ that F maps to an element, $\overline{E}$, that is not in $PointAngle_{2D}$. There are two ways that $\overline{E}$ can violate the requirements for membership in $PointAngle_{2D}$. First, $\overline{E}$'s θ_{PA} could be outside the interval $[0, 2\pi)$. Second, the point (x, y) could lie outside the unit circle. The first case is not possible, since E's θ_L is always within the approved interval and the transformation will not change θ_{PA} to be outside it. So our element $\overline{E}$ must lie outside the unit circle. But for E to be in $Lines_{2D}$, the location t on E's line must be in the unit circle. And our map goes to the same point in $PointAngle_{2D}$. So the mapping of E is in the unit circle. So we have reached a contradiction and we know that F is indeed a mapping from $Lines_{2D}$ to $PointAngle_{2D}$.

A.2.2 F forms a one-to-one correspondence

We will now demonstrate that F forms a one-to-one correspondence between $Lines_{2D}$ and $PointAngle_{2D}$. To show that it is a one-to-one correspondence, we must show that it is one-to-one and that it is onto.

To see that F is one-to-one, assume that $F(p_1, \theta_{L1}, t_1)$ is equal to $F(p_2, \theta_{L2}, t_2)$. We can give their common image a name, (x, y, θ_{PA}). Consider the following:

- The relationship between θ_L and θ_{PA} (equation A.20) shows that θ_{L1} must be equal to θ_{L2}.
- For a fixed θ_L, two distinct p's map to two disjoint chords on the circle. If the p's were different, but their image was at the same (x, y), then these disjoint chords must intersect. This can't happen, so the p's must also be equal.
- For a fixed p and θ_L, we can establish the point (x_1, y_1) and direction (x_d, y_d). t corresponds to the magnitude of offset of the point by the direction. The only way that that magnitude can vary and still end up at a common (x, y) is if the direction is the zero vector. This is not the case, so the t's must be equal.

So we have established that $(p_1, \theta_{L1}, t_1) = (p_2, \theta_{L2}, t_2)$, so the mapping must be one-to-one.

To see that F is onto, consider an element of $PointAngle_{2D}$, (x_0, y_0, θ_{PA0}). We will show that there exists a (p_0, θ_{L0}, t_0) that F maps to (x_0, y_0, θ_{PA0}). Again using equation A.20, we can immediately calculate θ_{L0}. Note that θ_{L0} will be in the range $[0, 2\pi)$. Further, we can calculate p_0 directly by subtituting values for θ_{L0}, x_0, and y_0 into A.2:

$$p_0 = x_0 \times sin(\theta_{L0}) + y_0 \times cos(\theta_{L0}) \tag{A.21}$$

To see that p_0 comes from the range $[-1, 1]$, we can use the following geometric identity:

$$x_0 \times sin(\theta_{L0}) + y_0 \times cos(\theta_{L0}) = \sqrt{x_0^2 + y_0^2} \times sin(\theta_{L0} + \beta) \tag{A.22}$$

where:

$$\beta = \begin{cases} arctan(y_0/x_0) & x_0 \geq 0 \\ arctan(y_0/x_0) \pm \pi & x_0 < 0 \end{cases} \tag{A.23}$$

Then $\sqrt{x_0^2 + y_0^2}$ is less than 1 (since (x_0, y_0) is contained within the unit circle) and $sin(\theta_{L0} + \beta)$ is bounded between -1 and 1, so their product, p_0, must come from the range $[-1, 1]$.

Finally, if $sin(\theta_{L0})$ is not 0, then we can solve for t_0 directly using equation A.18. If it is 0, then we can solve for t_0 using equation A.19.

Then we have shown that for an arbitrary (x_0, y_0, θ_{PA0}), we can construct a valid (p_0, θ_{L0}, t_0). Thus, F is onto.

A.2.3 The Jacobian of F

The Jacobian of F is the matrix of partial derivatives of F.

$$J(F) = \begin{vmatrix} \frac{\partial x}{\partial p} & \frac{\partial x}{\partial \theta_L} & \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial p} & \frac{\partial y}{\partial \theta_L} & \frac{\partial y}{\partial t} \\ \frac{\partial \theta_{PA}}{\partial p} & \frac{\partial \theta_{PA}}{\partial \theta_L} & \frac{\partial \theta_{PA}}{\partial t} \end{vmatrix} \tag{A.24}$$

and:

$$\frac{\partial x}{\partial p} = cos(\theta_L) + \frac{p \times sin(\theta_L)}{\sqrt{1 - p^2}} \tag{A.25}$$

$$\frac{\partial x}{\partial \theta_L} = (1 - \sqrt{1 - p^2}) \times cos(\theta_L) - p \times sin(\theta_L) \tag{A.26}$$

$$\frac{\partial x}{\partial t} = sin(\theta_L) \tag{A.27}$$

$$\frac{\partial y}{\partial p} = sin(\theta_L) - \frac{p \times cos(\theta_L)}{\sqrt{1-p^2}} \tag{A.28}$$

$$\frac{\partial y}{\partial \theta_L} = (1 - \sqrt{1-p^2}) \times sin(\theta_L) + p \times cos(\theta_L) \tag{A.29}$$

$$\frac{\partial y}{\partial t} = -cos(\theta_L) \tag{A.30}$$

$$\frac{\partial \theta_{PA}}{\partial p} = 0 \tag{A.31}$$

$$\frac{\partial \theta_{PA}}{\partial \theta_L} = 1 \tag{A.32}$$

$$\frac{\partial \theta_{PA}}{\partial t} = 0 \tag{A.33}$$

The determinant of J(F) is 1.

A.3 Three dimensional lines

The three dimensional directed line segment construction calls for first choosing a random point, P, along the surface of the sphere. $\overrightarrow{OP}$, the ray between the origin (O) and P, sets the direction of the line. A plane, T, is constructed. T has normal $\overrightarrow{OP}$ and intersects the point O. The intersection of the sphere with the plane is a disc. A random point, R, is selected from this disc. The final directed line is the line that goes through R and goes in direction $\overrightarrow{OP}$.

A.3.1 The mapping function F

The total number of random values needed to construct the directed line is four. Selecting a random point on the surface of a sphere requires two random values and selecting a point inside the disc (that the line is incident to) requires another two random values.

If P_0 is a random value from the range $(-1, 1)$ and θ_L is a random value from the range $[0, 2\pi)$, then a random point along the surface of the sphere can be constructed as:

$$x = cos(\theta_L) \times \sqrt{1 - P_0^2} \tag{A.34}$$

$$y = sin(\theta_L) \times \sqrt{1 - P_0^2} \tag{A.35}$$

$$z = P_0 \tag{A.36}$$

Note that this construction draws P_0 from an open interval and not a closed interval. As noted in section A.1, this will allow the mapping function we consider to be one-to-one which in turn will allow us to utilize it in theorem A.1.

Then the direction of the normal vector, N, for plane T can be immediately determined as (x, y, z).

Now consider the disc of intersection between T and the sphere. We will construct a disc for the plane "$Z = 0$" and then rotate that plane to be normal to N. We will use two random values, P_1 and P_2. Both are drawn uniformly from the interval $[-1, 1]$. P_1 and P_2 get mapped to the point $(P_1, P_2, 0)$ on the disc. If this point lies outside the disc, then new values of P_1 and P_2 are selected.

The plane will be rotated into position by composing two separate rotations. First, note that the angle ϕ_L can be calculated as $ArcCos(z)$. Then, the first rotation will be by ϕ_L around the y-axis. The second will be by θ_L around the z-axis. The first rotation is:

$$R_{y;\phi_L} = \begin{vmatrix} cos(\phi_L) & 0 & -sin(\phi_L) \\ 0 & 1 & 0 \\ sin(\phi_L) & 0 & cos(\phi_L) \end{vmatrix} \tag{A.37}$$

The second rotation is:

$$R_{z;\theta_L} = \begin{vmatrix} cos(\theta_L) & sin(\theta_L) & 0 \\ -sin(\theta_L) & cos(\theta_L) & 0 \\ 0 & 0 & 1 \end{vmatrix} \tag{A.38}$$

The composition of the matrices is:

$$R = \begin{vmatrix} cos(\theta_L) \times cos(\phi_L) & sin(\theta_L) \times cos(\phi_L) & -sin(\phi_L) \\ -sin(\theta_L) & cos(\theta_L) & 0 \\ cos(\theta_L) \times sin(\phi_L) & sin(\theta_L) \times sin(\phi_L) & cos(\phi_L) \end{vmatrix} \tag{A.39}$$

And a point P on the disc becomes:

$$(P1, P2, 0) \times R = (x_0, y_0, z_0) \tag{A.40}$$

where:

$$x_0 = cos(\theta_L) \times cos(\phi_L) \times P_1 - sin(\theta_L) \times P_2 \tag{A.41}$$

$$y_0 = sin(\theta_L) \times cos(\phi_L) \times P_1 + cos(\theta_L) \times P_2 \tag{A.42}$$

$$z_0 = -sin(\phi_L) \times P_1 \tag{A.43}$$

We are now ready to construct the mapping $F(P_0, \theta_L, P_1, P_2, t) \rightarrow (x, y, z, \theta_{PA}, \phi_{PA})$. We have previously introduced the parameters for constructing a random directed line: P_0, θ_L, P_1, and P_2. The final parameter, t, is the distance along that line. Note that, unlike the two dimensional case, t does not represent a distance from one point of the surface of the sphere to another. Instead, t represents the distance from the point (x_0, y_0, z_0). The only valid values of t are from the interval $[-\sqrt{1 - P_1^2 - P_2^2}, +\sqrt{1 - P_1^2 - P_2^2}]$. They correspond to the points along the line within the unit sphere.

Again we would like to determine functions of the form:

$$x = F_x(P_0, \theta_L, P_1, P_2, t) \tag{A.44}$$

$$y = F_y(P_0, \theta_L, P_1, P_2, t) \tag{A.45}$$

$$z = F_z(P_0, \theta_L, P_1, P_2, t) \tag{A.46}$$

$$\theta_{PA} = F_{\theta_{PA}}(P_0, \theta_L, P_1, P_2, t) \tag{A.47}$$

$$\phi_{PA} = F_{\phi_{PA}}(P_0, \theta_L, P_1, P_2, t) \tag{A.48}$$

With equations A.41, A.42, and A.43, we established a point on our random directed line. And with A.34, A.35, A.36, we established the direction of that line. Then F_x, F_y, and F_z are straight-forward to establish.

$$F_x(P_0, \theta_L, P_1, P_2, t) = x_0 + t \times N_x \tag{A.49}$$

$$F_y(P_0, \theta_L, P_1, P_2, t) = y_0 + t \times N_y \tag{A.50}$$

$$F_z(P_0, \theta_L, P_1, P_2, t) = z_0 + t \times N_z \tag{A.51}$$

$F_{\theta_{PA}}$ and $F_{\phi_{PA}}$ are trivial. Their directions align with the directed line.

$$F_{\theta_{PA}}(P_0, \theta_L, P_1, P_2, t) = \theta_L \tag{A.52}$$

$$F_{\phi_{PA}}(P_0, \theta_L, P_1, P_2, t) = \phi_L = ArcCos(P_0) \tag{A.53}$$

Again, we must argue that F only maps to valid tuples $(x, y, z, \theta_{PA}, \phi_{PA})$ to show that F is a map from $Lines_{3D}$ to $PointAngle_{3D}$. Assume F maps $(P_0, \theta_L, P_1, P_2, t)$ to an invalid $(x, y, z, \theta_{PA}, \phi_{PA})$. From equations A.52 and A.53, we can directly infer the angles and determine that they are valid. So that means that (x, y, z) must be outside the unit sphere. The directed line formed by $(P_0, \theta_L, P_1, P_2)$ must intersect the sphere (to form a chord) due to the nature of its construction. So we must have a value of t that does not lie on this chord. But t is limited to the portions that are along the chord. So we have reached a contradiction and we know that F is a valid mapping from $Lines_{3D}$ to $PointAngle_{3D}$.

A.3.2 F forms a one-to-one correspondence

We will now demonstrate that our mapping F for three dimensional directed lines forms a one-to-one correspondence between $Lines_{3D}$ and $PointAngle_{3D}$ over the open subsets we are considering.

To see that F is one-to-one, assume that $F(P_{0,1}, \theta_{L1}, P_{1,1}, P_{2,1}, t_1)$ is equal to $F(P_{0,2}, \theta_{L2}, P_{1,2}, P_{2,2}, t_2)$. For ease of reference name their image $(x, y, z, \theta_{PA}, \phi_{PA})$. Consider the following:

- Equations A.52 and A.53 immediately show that $P_{0,1} = P_{0,2}$ and $\theta_{L1} = \theta_{L2}$.
- For a fixed P_0 and θ_L, each (P_1, P_2) pair maps to a unique ray[2]. Since the rays are parallel, they do not intersect. We know that F maps the two tuples to the same (x, y, z), so the rays they lie on must intersect. Hence $P_{1,1} = P_{1,2}$ and $P_{2,1} = P_{2,2}$.
- For the tuples to map to the same location (x, y, z), they must be offset from the same origin along the normal direction by the same amount. Hence $t_1 = t_2$.

So we have established that $F(P_{0,1}, \theta_{L1}, P_{1,1}, P_{2,1}, t_1) = F(P_{0,2}, \theta_{L2}, P_{1,2}, P_{2,2}, t_2)$, so the mapping must be one-to-one.

To see that F is onto, consider an element of $PointAngle$, $(x, y, z, \theta_{PA}, \phi_{PA})$. Note that ϕ_{PA} cannot be 0 or π since we are considering only the open subset. We will show that there exists a $(P_0, \theta_L, P_1, P_2, t)$, with P_0 not equal to -1 or 1, that has this image

[2]Note that this property is not true if we allow P_0 to be -1 or 1, since $P_0 = -1$ always maps to the ray $(0, 0, 0) \rightarrow (0, 0, -1)$ and $P_0 = 1$ always maps to the ray $(0, 0, 0) \rightarrow (0, 0, 1)$ for any choice of θ.

through F. Again using equations A.52 and A.53, we can immediate calculate P_0 and θ_L. Since ϕ_{PA} does not equal 0 or π, P_0 will be within $(-1, 1)$. Then we can use P_0 and θ_L to establish the normal direction. Next we start at point (x, y, z) and move along the normal direction until it hits the the disc created by intersecting the sphere with the plane Q. The distance we move is t. We know that this process (moving in the direction of the normal) will ultimately lead to an intersection with the disc, because extruding along the disc in the direction of the normal will create an infinite cylinder that fully contains the sphere. If the point were not within this cylinder, it would lie outside the sphere, which would mean that $(x, y, z, \theta_{PA}, \phi_{PA})$ would not be a valid member of $PointAngle_{3D}$. Finally, given the point of intersection with the disc, we can calculate P_1 and P_2 by multiplying the point with the inverse of the rotation matrix from A.39. Since we were able to prove the existence of an element from $Lines_{3D}$, F is onto.

A.3.3 The Jacobian of F

Again, we will calculate the Jacobian of F, the matrix of its partial derivatives.

$$J(F) = \begin{vmatrix} \frac{\partial x}{\partial P_0} & \frac{\partial x}{\partial \theta_L} & \frac{\partial x}{\partial P_1} & \frac{\partial x}{\partial P_2} & \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial P_0} & \frac{\partial y}{\partial \theta_L} & \frac{\partial y}{\partial P_1} & \frac{\partial y}{\partial P_2} & \frac{\partial y}{\partial t} \\ \frac{\partial z}{\partial P_0} & \frac{\partial z}{\partial \theta_L} & \frac{\partial z}{\partial P_1} & \frac{\partial z}{\partial P_2} & \frac{\partial z}{\partial t} \\ \frac{\partial \theta_{PA}}{\partial P_0} & \frac{\partial \theta_{PA}}{\partial \theta_L} & \frac{\partial \theta_{PA}}{\partial P_1} & \frac{\partial \theta_{PA}}{\partial P_2} & \frac{\partial \theta_{PA}}{\partial t} \\ \frac{\partial \phi_{PA}}{\partial P_0} & \frac{\partial \phi_{PA}}{\partial \theta_L} & \frac{\partial \phi_{PA}}{\partial P_1} & \frac{\partial \phi_{PA}}{\partial P_2} & \frac{\partial \phi_{PA}}{\partial t} \end{vmatrix} \tag{A.54}$$

and:

$$\frac{\partial x}{\partial P_0} = cos(\theta_L) \times (P_1 - \frac{t \times P_0}{\sqrt{1 - P_0^2}}) \tag{A.55}$$

$$\frac{\partial x}{\partial \theta_L} = -P_2 \times cos(\theta_L) - sin(\theta_L) \times (P_0 \times P_1 + t \times \sqrt{1 - P_0^2}) \tag{A.56}$$

$$\frac{\partial x}{\partial P_1} = P_0 \times cos(\theta_L) \tag{A.57}$$

$$\frac{\partial x}{\partial P_2} = -sin(\theta_L) \tag{A.58}$$

$$\frac{\partial x}{\partial t} = \sqrt{1 - P_0^2} \times cos(\theta_L) \tag{A.59}$$

$$\frac{\partial y}{\partial P_0} = sin(\theta_L) \times (P_1 - \frac{t \times P_0}{\sqrt{1 - P_0^2}}) \tag{A.60}$$

$$\frac{\partial y}{\partial \theta_L} = -P_2 \times sin(\theta_L) + cos(\theta_L) \times (P_0 \times P_1 + t \times \sqrt{1 - P_0^2}) \tag{A.61}$$

$$\frac{\partial y}{\partial P_1} = P_0 \times sin(\theta_L) \tag{A.62}$$

$$\frac{\partial y}{\partial P_2} = cos(\theta_L) \tag{A.63}$$

$$\frac{\partial y}{\partial t} = \sqrt{1 - P_0^2} \times sin(\theta_L) \tag{A.64}$$

$$\frac{\partial z}{\partial P_0} = t + \frac{P_0 \times P_1}{\sqrt{1 - P_0^2}} \tag{A.65}$$

$$\frac{\partial z}{\partial \theta_L} = 0 \tag{A.66}$$

$$\frac{\partial z}{\partial P_1} = -\sqrt{1 - P_0^2} \tag{A.67}$$

$$\frac{\partial z}{\partial P_2} = 0 \tag{A.68}$$

$$\frac{\partial z}{\partial t} = P_0 \tag{A.69}$$

$$\frac{\partial \theta_{PA}}{\partial P_0} = 0 \tag{A.70}$$

$$\frac{\partial \theta_{PA}}{\partial \theta_L} = 1 \tag{A.71}$$

$$\frac{\partial \theta_{PA}}{\partial P_1} = 0 \tag{A.72}$$

$$\frac{\partial \theta_{PA}}{\partial P_2} = 0 \tag{A.73}$$

$$\frac{\partial \theta_{PA}}{\partial t} = 0 \tag{A.74}$$

$$\frac{\partial \phi_{PA}}{\partial P_0} = -\frac{1}{\sqrt{1 - P_0^2}} \tag{A.75}$$

$$\frac{\partial \phi_{PA}}{\partial \theta_L} = 0 \tag{A.76}$$

$$\frac{\partial \phi_{PA}}{\partial P_1} = 0 \tag{A.77}$$

$$\frac{\partial \phi_{PA}}{\partial P_2} = 0 \tag{A.78}$$

$$\frac{\partial \phi_{PA}}{\partial t} = 0 \tag{A.79}$$

The determinant of J(F) is $\frac{1}{\sqrt{1-P_0^2}}$. Note that $\sqrt{1 - P_0^2}$ is $sin(\phi_{PA})$, so the determinant is $\frac{1}{sin(\phi_{PA})}$.

Appendix B

Calculating volume and mass as a function of length scale

The definitions of mass as a function of length scale from chapter 8 are computationally expensive; they require examining all points in the volume and all chords through those points. Our approach for calculating these quantities was instead to cast uniform density, random directed lines and perform analysis on the resulting intersections. We will now demonstrate this technique will converge to the calculation of the previously defined quantities.

B.1 Volume Below α

Consider the effect of casting N uniform density, random directed lines. Each directed line is intersected with the shape (Q) to produce segments. The lengths of the segments are then determined. There are two counters, CNT_1 and CNT_2. CNT_1 is the total length encountered from segments with length below α. CNT_2 is the total length encountered for all segments. Denote the total volume of the object, Q, as V_T and denote the volume at length scale less than α as $V_{le}(\alpha)$. Then we claim:

$$V_{le}(\alpha) = V_T \times \lim_{N\to\infty} \frac{CNT_1}{CNT_2} \tag{B.1}$$

To help prove this, consider a helper function, $D(m, \alpha)$. $D(m, \alpha)$ is the total

lengths of segments from the directed line m that have length less than or equal to α. So, if a particular line scan, m_0, contains segments of lengths 2, 3, and 5, then $D(m_0, 2.5) = 2$, while $D(m_0, 4) = 5$ and $D(m_0, 1) = 0$. Similarly, if we wanted to have a function that counted the lengths of all the segments encountered on m_0, we could use $D(m_0, \infty)$.

Consider the average value of $\frac{D(m,\alpha)}{D(m,\infty)}$, which we will refer to as $\overline{D(\alpha)}$. This average can be constructed by considering this ratio for all possible lines that go through the shape Q. If "dm" is the variable of integration over our random, uniform density directed line construction and MQ is the set of directed lines that intersect the sphere circumscribing Q, then:

$$\overline{D(\alpha)} = \frac{\int_{MQ} D(m, \alpha) dm}{\int_{MQ} D(m, \infty) dm} \tag{B.2}$$

Let's now consider the ratio of CNT_1 and CNT_2 again. For each directed line m considered, the contribution to CNT_1 is the sum of the lengths of the segments that are less than α. Our helper function, $D(m, \alpha)$, is counting the exact same thing. Similarly, the function $D(m, \infty)$ is counting the same thing as CNT_2. Then the law of large numbers states that if we consider more and more directed lines, then our ratio will reach the average, which is $\overline{D(\alpha)}$:

$$\lim_{N \to \infty} \frac{CNT_1}{CNT_2} = \overline{D(\alpha)} \tag{B.3}$$

Now let's consider $\overline{D(\alpha)}$ in more detail. Our helper function $D(m, \alpha)$ returns lengths of segments. But the length of a segment is merely an integration along that segment. We can capture this notion by using a new helper function, $D_{point}(m, t, \alpha)$. t is the distance along m, so (m, t) corresponds to some point P. Then $D_{point}(m, t, \alpha)$ returns 1 if P is contained by a segment that has length less than or equal to α. Otherwise, $D_{point}(m, t, \alpha)$ returns 0. (Note that $D_{point}(m, t, \alpha)$ is the *Lines* equivalent of $S_{le}(P, \theta, \alpha)$ for the set *PointAngle*.) So, if "dt" is the variable of integration along a given directed line m, then:

$$D(m, \alpha) = \int_{PQm} D_{point}(m, t, \alpha) dt \tag{B.4}$$

where PQm is the set of points P in the shape Q that lie along the directed line m. Then,

substituting equation B.4 into equation B.2 gives:

$$\overline{D(\alpha)} = \frac{\int_{MQ}\int_{PQm} D_{point}(m,t,\alpha)dtdm}{\int_{MQ}\int_{PQm} D_{point}(m,t,\infty)dtdm} \tag{B.5}$$

As discussed in section A.1, a theorem links evaluations from one set with evaluations on another. For our mapping function, F, the sets $Lines$ and $PointAngle$, and an arbitrary function h defined on $PointAngle$:

$$\int_{PointAngle} h(a)dA = \int_{Lines} h(F(b)) \times J_F(b)dB \tag{B.6}$$

where $J_F(b)$ is the determinant of the Jacobian of F at point b.

The function, h, will will consider is S_{le}. For two dimensions, $h(a)$ is $S_{le}(P,\theta,\alpha)$. For three dimensions, $h(a)$ includes the $sin(\phi)$ term that prevents biasing. In that case, $h(a)$ is $S_{le}(P,(\theta,\phi),\alpha) \times sin(\phi)$. Then:

$$\int_Q \int_0^{2\pi} S_{le}(P,\theta,\alpha)d\theta dV = \int_{MQ}\int_{PQm} S_{le}(F(m,t),\alpha) \times J_F(m,t)dtdm \quad [2D] \tag{B.7}$$

$$\int_Q \int_0^{2\pi} \int_0^{\pi} S_{le}(P,(\theta,\phi),\alpha) \times sin(\phi)d\phi d\theta dV =$$

$$\int_{MQ}\int_{PQm} S_{le}(F(m,t),\alpha) \times sin(\phi) \times J_F(m,t)dtdm \quad [3D] \tag{B.8}$$

The ϕ term in the right hand side of equation B.8 actually corresponds to $F_\phi(m,t)$, from equation A.53, which maps (m,t) to ϕ in the set $PointAngle$.

Note that the constructions of D_{point} and S_{le} and the mapping function F give us a nice property. $D_{point}(m,t,\alpha)$ is equal to $S_{le}(F(m,t),\alpha)$ for any m and t. So:

$$\int_Q \int_0^{2\pi} S_{le}(P,\theta,\alpha)d\theta dV = \int_{MQ}\int_{PQm} D_{point}(m,t,\alpha) \times J_F(m,t)dtdm \quad [2D] \tag{B.9}$$

$$\int_Q \int_0^{2\pi} \int_0^{\pi} S_{le}(P,(\theta,\phi),\alpha) \times sin(\phi)d\phi d\theta dV =$$

$$\int_{MQ}\int_{PQm} D_{point}(m,t,\alpha) \times sin(\phi) \times J_F(m,t)dtdm \quad [3D] \tag{B.10}$$

We showed in Appendix A that the determinant of the Jacobian for the two dimensional case was 1, so it can be dropped from the equation.

$$\int_Q \int_0^{2\pi} S_{le}(P,\theta,\alpha)d\theta dV = \int_{MQ}\int_{PQm} D_{point}(m,t,\alpha)dtdm \quad [2D] \tag{B.11}$$

In three dimensions, we showed that the determinant of the Jacobian was $\frac{1}{sin(\phi)}$. This cancels with the $sin(\phi)$ from S_{le}'s three dimensional definition. This gives:

$$\int_Q \int_0^{2\pi} \int_0^{\pi} S_{le}(P,(\theta,\phi),\alpha) \times sin(\phi) d\phi d\theta dV = \int_{MQ} \int_{PQm} D_{point}(m,t,\alpha) dt dm \ [3D] \quad \text{(B.12)}$$

Recall that in section A.1, we stated that the integrand over our excluded region is finite. As D_{point} is always 0 or 1, we can now see that that is true.

We can now unify our two and three dimensional forms and return to the shorthand where ω represents θ in two dimensions and (θ,ϕ) in three dimensions:

$$\int_Q \int S_{le}(P,\omega,\alpha) d\omega dV = \int_{MQ} \int_{PQm} D_{point}(m,t,\alpha) dt dm \quad \text{(B.13)}$$

Similarly, for α equal to ∞:

$$\int_Q \int S_{le}(P,\omega,\infty) d\omega dV = \int_{MQ} \int_{PQm} D_{point}(m,t,\infty) dt dm \quad \text{(B.14)}$$

Further, $S_{le}(P,\omega,\infty)$ always returns 1 for any point P in Q, so:

$$\int_Q \int 1 d\omega dV = \int_{MQ} \int_{PQm} D_{point}(m,t,\infty) dt dm \quad \text{(B.15)}$$

Taking the ratio from B.13 and B.15 gives us a form that can be related to B.5:

$$\frac{\int_{MQ} \int_{PQm} D_{point}(m,t,\alpha) dt dm}{\int_{MQ} \int_{PQm} D_{point}(m,t,\infty) dt dm} = \frac{\int_Q \int S_{le}(P,\omega,\alpha) d\omega dV}{\int_Q \int 1 d\omega dV} \quad \text{(B.16)}$$

Summarizing the results from equations B.2, B.3, B.5, and B.16, we can see the relationship between the ratio we started with and our definition:

$$\lim_{N\to\infty} \frac{CNT_1}{CNT_2} = \overline{D(\alpha)} = \frac{\int D(m,\alpha) dm}{\int D(m,\infty) dm} =$$

$$\frac{\int_{MQ} \int_{PQm} D_{point}(m,t,\alpha) dt dm}{\int_{MQ} \int_{PQm} D_{point}(m,t,\infty) dt dm} = \frac{\int_Q \int S_{le}(P,\omega,\alpha) d\omega dV}{\int_Q \int 1 d\omega dV} \quad \text{(B.17)}$$

We are really only interested in the first and last quantities in this relationship:

$$\lim_{N\to\infty} \frac{CNT_1}{CNT_2} = \frac{\int_Q \int S_{le}(P,\omega,\alpha) d\omega dV}{\int_Q \int 1 d\omega dV} \quad \text{(B.18)}$$

Because $\int 1 d\omega$ is a constant, this constant can be pulled out of the integral in the denominator, placed in the numerator (as $\frac{1}{\int 1 d\omega}$) and then placed inside the numerator's integral.

$$\lim_{N\to\infty} \frac{CNT_1}{CNT_2} = \frac{\int \frac{\int S_{le}(P,\omega,\alpha) d\omega}{\int 1 d\omega} dV}{\int 1 dV} \quad \text{(B.19)}$$

Substituting in equation 8.2 gives:

$$\lim_{N\to\infty}\frac{CNT_1}{CNT_2}=\frac{\int C_P(\alpha)dV}{\int 1dV} \tag{B.20}$$

And using that $\int 1dV$ is V_T gives:

$$\lim_{N\to\infty}\frac{CNT_1}{CNT_2}=\frac{\int C_P(\alpha)dV}{V_T} \tag{B.21}$$

Cross multiplying (and re-arranging) gives:

$$\int C_P(\alpha)dV = V_T\times\lim_{N\to\infty}\frac{CNT_1}{CNT_2} \tag{B.22}$$

And using the definition of $V_{le}(\alpha)$ (equation 8.4) gives the final result:

$$V_{le}(\alpha) = V_T\times\lim_{N\to\infty}\frac{CNT_1}{CNT_2} \tag{B.23}$$

B.2 Mass Below α

The mass below α is calculated in a similar way to the volume. The counters, CNT_1 and CNT_2 are updated with $\int\rho(P)dL$ over the chords. Although this quantity would appear to create some sort of "linear mass", the correct interpretation of this is as a portion of a larger $\int\rho(P)dV$ integration with the points chosen for this part of the integral that coincidentally lie along a line.

Rather than considering $\overline{D(\alpha)}$, we will now consider $\overline{D_{mass}(\alpha)}$:

$$\overline{D_{mass}(\alpha)}=\frac{\int_{MQ}\int_{PQm}\rho(P)\times D_{point}(m,t,\alpha)dtdm}{\int_{MQ}\int_{PQm}\rho(P)\times D_{point}(m,t,\infty)dtdm} \tag{B.24}$$

Because CNT_1 and CNT_2 now consider density weights, $\overline{D_{mass}(\alpha)}$ will still mirror these counters. Then, skipping the redundant portions of the discussion from section B.1, we get:

$$\lim_{N\to\infty}\frac{CNT_1}{CNT_2}=\frac{\int_Q\int\rho(P)\times S_{le}(P,\omega,\alpha)d\omega dV}{\int_Q\int\rho(P)d\omega dV} \tag{B.25}$$

Once again pulling out the constant $\int 1d\omega$ and placing it in the numerator and then substituting in the definition of $C_P(\alpha)$ gives:

$$\lim_{N\to\infty}\frac{CNT_1}{CNT_2}=\frac{\int_Q\rho(P)C_P(\alpha)dV}{\int_Q\rho(P)dV} \tag{B.26}$$

$\int \rho(P)dV$ is the mass of the object (M_T) and $\int \rho(P)C_P(\alpha)dV$ is the mass at length scale below α ($M_{le}(\alpha)$, from equation 8.6), so we get a similar result:

$$M_{le}(\alpha) = M_T \times \lim_{N\to\infty} \frac{CNT_1}{CNT_2} \tag{B.27}$$

B.3 Distribution of Volume By Scale

To calculate the distribution, we make counters $CNT(\alpha, \Delta)$. These counters store the total length of all the chords that have length in the interval $[\alpha, \alpha + \Delta)$.

Our goal is calculate the function $V_{eq}(\alpha, \Delta)$ from 8.1.2. We will show that if V_T is the total volume, then, as the number of directed lines approaches infinity:

$$V_{eq}(\alpha, \Delta) = V_T \times \frac{CNT(\alpha, \Delta)}{\sum_{n=0}^{\infty} CNT(n \times \Delta, \Delta)} \tag{B.28}$$

Now we will consider a new function, $D_R(m, \alpha, \Delta)$, which sums the lengths of segments in the range $[\alpha, \alpha+\Delta)$. (The 'R' is for "Range".) So, if a particular line scan, m_0, produces segments of lengths 2, 3, and 5, then $D_R(m_0, 2.5, 1) = 3$, while $D_R(m_0, 2.5, 0.4) = 0$ and $D_R(m_0, 1.5, 2) = 5$. Similarly, if we wanted to have a function that counted the lengths of all the segments on m_0, we could use $D_R(m_0, 0, \infty)$.

Consider the average value of $\frac{D_R(m,\alpha,\Delta)}{D_R(m,0,\infty)}$, which we will refer to as $\overline{D_R(\alpha, \Delta)}$. This average can be constructed by considering this ratio for all possible directed lines that go through the shape Q. If "dm" is the variable of integration over our random, uniform density directed lines, then:

$$\overline{D_R(\alpha, \Delta)} = \frac{\int_{MQ} D_R(m, \alpha, \Delta)dm}{\int_{MQ} D(m, 0, \infty)dm} \tag{B.29}$$

where MQ is the set of all random, uniform density directed lines that intersect the sphere circumscribing Q. Note that this is the same as considering only the lines that actually intersect Q, since the lines that do not intersect Q, but do intersect its bounding sphere, only contribute 0's.

Let's now consider the ratio of $CNT(\alpha, \Delta)$ and $\sum_{n=0}^{\infty} CNT(n \times \Delta, \Delta)$. For each directed line m considered, the contribution to $CNT(\alpha, \Delta)$ is the sum of the lengths of

the segments that are in the range $[\alpha, \alpha + \Delta)$. Our helper function $D_R(m, \alpha, \Delta)$ is again counting this exact same thing. Similarly, the function $D(m, 0, \infty)$ is counting the same thing as $\sum_{n=0}^{\infty} CNT(n \times \Delta, \Delta)$. Then the law of large numbers states that if we consider more and more directed lines, then our ratio will reach the average, which is $\overline{D_R(\alpha, \Delta)}$:

$$\lim_{N \to \infty} \frac{CNT(\alpha, \Delta)}{\sum_{n=0}^{\infty} CNT(n \times \Delta, \Delta)} = \overline{D_R(\alpha, \Delta)} \tag{B.30}$$

Now let's consider $\overline{D_R(\alpha, \Delta)}$ in more detail. Our helper function $D_R(m, \alpha, \Delta)$ returns lengths of segments. But the length of a segment is merely an integration along that segment. We can capture this notion by introducing another helper function, $D_{R,point}(m, t, \alpha, \Delta)$. The new t parameter corresponds to a distance along m. Going this distance along m yields a point in the shape Q. If the directed line m intersected with Q forms a segment that contains this point and the length of the segment is between α and $\alpha + \Delta$, then $D_{R,point}(m, t, \alpha, \Delta)$ returns 1. Otherwise it returns 0. (Note that $D_{R,point}(m, t, \alpha, \Delta)$ is the *Lines* equivalent of $S_{eq}(P, \omega, \alpha, \Delta)$ for the set $PointAngle$.) So, if "dt" is the variable of integration along a given directed line m, then:

$$D_R(m, \alpha, \Delta) = \int_{PQm} D_{R,point}(m, t, \alpha, \Delta) dt \tag{B.31}$$

where PQm is the set of points P in the shape Q that lie along m. Then, substituting equation B.31 into equation B.29 gives:

$$\overline{D_R(\alpha, \Delta)} = \frac{\int \int D_{R,point}(m, t, \alpha, \Delta) dt dm}{\int \int D_{R,point}(m, t, 0, \infty) dt dm} \tag{B.32}$$

We can again use the theorem relating results over two sets by a magnification of the Jacobian of their mapping function. Skipping redundant steps from section B.1 (with $D_{R,point}$ playing the role of D_{point} and S_{eq} playing the role of S_{le}), we find:

$$\lim_{N \to \infty} \frac{CNT(\alpha, \Delta)}{\sum_{n=0}^{\infty} CNT(n \times \Delta, \Delta)} = \frac{\int_Q \int S_{eq}(P, \omega, \alpha, \Delta) d\omega dV}{\int_Q \int 1 d\omega dV} \tag{B.33}$$

Again, we can remove the constant $\int 1 d\omega$ from the denominator and place it in the numer-

ator, to obtain:

$$\lim_{N\to\infty} \frac{CNT(\alpha,\Delta)}{\sum_{n=0}^{\infty} CNT(n\times\Delta,\Delta)} = \frac{\int_Q \frac{\int S_{eq}(P,\omega,\alpha,\Delta)d\omega}{\int 1 d\omega} dV}{\int_Q 1 dV} \tag{B.34}$$

And then substitute in the definition of $D_P(\alpha,\Delta)$ from 8.9 to get:

$$\lim_{N\to\infty} \frac{CNT(\alpha,\Delta)}{\sum_{n=0}^{\infty} CNT(n\times\Delta,\Delta)} = \frac{\int_Q D_P(\alpha,\Delta)dV}{\int_Q 1 dV} \tag{B.35}$$

Substituting $\int 1dV = V_T$ and cross multiplying gives:

$$V_{eq}(\alpha,\Delta) = V_T \times \lim_{N\to\infty} \frac{CNT(\alpha,\Delta)}{\sum_{n=0}^{\infty} CNT(n\times\Delta,\Delta)} \tag{B.36}$$

From a practical sense, we can obtain the volume at length scales $[\alpha, \alpha+\Delta)$ by dividing the counter for those length scales by the sum of the counters at all other length scales and multiplying by the total volume.

B.4 Distribution of Mass By Scale

This derivation is very similar to the previous three. It incorporates the "mass" elements from section B.2 and the "distribution" elements from section B.3. It yields the result:

$$M_{eq}(\alpha,\Delta) = M_T \times \lim_{N\to\infty} \frac{CNT(\alpha,\Delta)}{\sum_{n=0}^{\infty} CNT(n\times\Delta,\Delta)} \tag{B.37}$$

where M_T is the total mass and $CNT(\alpha,\Delta)$ is the counter for how much mass was encountered at length scale α when casting directed lines.